University of Pisa

Faculty of Engineering

Second level degree in Electronic Engineering

Curriculum in Electronic Systems

Development and Testing of an FPGA Prototype of Cryptographic Modules for Mobile Phone Security

Internship project with:

INFINEON Technologies France

Student:

Enrico Papa

University Supervisor :

Luca Fanucci

Infineon Tutors:

Simone Borri

Christophe Lopez

Contents

A Mamma, Papà e Laura

Acknowledgements

I would like to thank my University Supervisor, Luca FANUCCI, and INFINEON for giving me an interesting stage opportunity for my formation.

I also would like to thank my INFINEON's tutors, Simone BORRI and Christophe LOPEZ, for having followed and helped me during my stage. I am grateful to them for their time dedicated in teaching me and helping me solve my problems, even if I knew they were often busy with their company work.

1. Introduction

1.1 Infineon

Figure 1.1: Infineon's logo

Infineon was founded by Siemens in 1999 in order to take over its semiconductors sector. Today, it counts more than 40.000 employees around the world, 7.100 of which are involved in research and development.

In May 2006, Infineon's Memory Products Division was spun off to a distinct company called Qimonda. It employs about 12.000 people worldwide. Infineon was among the top 6 worldwide semiconductors sales in 2005, and now, even without the memory sector still lies among the top 20.
Infineon is present around the world through its Research & Development centers and foundries in North America (USA, Canada), Asia (China, Singapore, Taiwan, Japan) and Europe (Germany, France, UK, Austria, Romania).
Infineon has two main sectors of activity. The first one is the Automotive, Industrial, and Multi-market (AIM). It offers a wide range of products going from Engine Control Units (ECU) to ABS through airbag and air-conditioning control. As a result of many years of experience in the automotive field, Infineon has acquired a very high standard of application know-how and offers a comprehensive product portfolio of sensors, microcontrollers and power ICs among others.

Its second main sector of activity is the Communications sector (COM). Infineon business group develops, manufactures and markets end-to-end leading edge semiconductor products and solutions for cellular, wireless and wired communications enabling smooth voice and data high speed transmission from the backbone of the telecommunication network infrastructures to the end user's equipment.

The Research and Development center of Sophia Antipolis has about 140 employees. The development teams are working on three main development activities:

- **Mobile Solutions**

The mobile communications group is responsible for specifying and designing baseband integrated circuits for mobile phones (GSM/GPRS,EDGE). The teams specialize in algorithm development, system specification and validation, circuit design and embedded software. Since 2005, the development team specializes in the baseband devices with integration of RF technology to serve the entry phone and ultra low cost market.

Figure 1.2 : Infineon's COM products and market position

- **Memories library development**

The library group uses its expertise in analog design and simulation, digital modelling, testing and mask design to develop libraries for single and dual port, high frequency, low power and high density embedded SRAM silicon memory compilers.

- **Design Flow software development**

The design flow software group develops software products for automation of IC design including analog/digital macro export, memory compiler tool suits, user friendly web interfaces and a design timing sign-off platform.

1.2 Internship subject description

Given the increasing complexity of applications managed by today's mobile phones/PDA, there is a growing need for embedded security in the mobile baseband processors arena. Standard cryptography algorithms including SHA-x hash functions, 3-DES/AES symmetric ciphers and asymmetric cryptosystems, e.g. RSA, are usually adopted for applications such as: secure-boot, digital signatures, SSL and Digital Rights Management (DRM) support.

Hardware cryptographic accelerators are often needed to reduce the computational burden on the on-chip microprocessor(s) and must be developed as reusable intellectual properties (IPs) blocks with the aim of serving a vaste range of applications while minimizing cost and power. As for all complex IPs, special care is needed in ensuring complete HW verification and proper interaction with real-time embedded SW drivers.

The internship has focused on realizing an FPGA prototype of a cryptographic IP including several cryptographic accelerator modules with the double aim of:

1. doing an high level verification; and
2. providing an early platform for development of embedded SW drivers.

It has consisted in the following steps:

- VHDL-RTL FPGA design description
- Simulation of the design with Modelsim
- Porting onto a prototyping platform based on an ARM9 + Xilinx VirtexII FPGA
- Application oriented verification via the ARM9 in C code.

1.3 Mobile Phone Security

Infineon must follow some security and integrity requirements for their mobile phone platforms. Those are, for example:

IMEI Protection : The International Mobile Equipment Identity (IMEI) is a number unique to every GSM and UMTS mobile phone. The IMEI number is used by the GSM network to identify valid devices and therefore can be used to stop a stolen phone from accessing the network. For example, if a mobile phone is stolen, the owner can call his or her network provider and instruct them to "blacklist" (ban) the phone using its IMEI number. This renders the phone useless, regardless of whether the phone's SIM is changed.

Sim Lock : A SIM lock is a capability built-in to GSM phones by mobile phone manufacturers. Network providers use this capability to restrict the use of these phones to specific countries and network providers. Currently, phones can be locked to accept only SIM cards from one or more of the following: Countries, Network/Service providers, SIM types. The most common lock is the service provider lock (SP-lock). That kind of lock ensures that the handset is only used with SIM cards for the same service provider that marketed the handset.

Digital rights management : Digital rights management (DRM) is a term that refers to access control technologies used by publishers and copyright holders to limit usage of digital media or devices.

Secure Sockets Layer : Secure Socket Layer (SSL) is a cryptographic protocols that provide secure communications on the Internet for such things as web browsing, e-mail, Internet faxing, instant messaging and other data transfers. The SSL protocol allows applications to communicate across a network in a way designed to prevent eavesdropping, tampering, and message forgery. TLS provides endpoint authentication and communications privacy over the Internet using cryptography

Secure Boot : The purpose of the secure boot is to perform an integrity check of the code in the program FLASH. This can be done for example to be sure that it comes from a trusted source. In the ROM, together with the code, a signature is also generated for an authentication purpose. To do this the first part of the code is first hashed and then encrypted with the owner’s secret key. At every boot procedure, that part of the code is hashed and the value is compared with the expected one. if they are not the same the boot fails.

Secure Flash update : The purpose of the secure code download process is to provide a mechanism by which authorized parties can install code updates into the non-volatile storage of the Mobile Equipment using its local interfaces. Typically serial ports or USB ports are used for this purpose. This mechanism could be used to recover terminals which have been rendered inoperable by

external failures (for example cosmic rays affecting the flash) or where new software versions are to be installed.

For example a mechanism by which this can be achieved is that at boot time the boot code detects from signals on the hardware interface that a secure code download is being requested. At this point a small program is downloaded over this interface which is checked for authenticity and integrity before it is executed. This program then carries out the remainder of the flash download process: it must verify the integrity and authenticity of all code which is downloaded.

1.4 Example of a mobile phone platform with crypto accelerator

In figure 1.3 it is illustrated an architecture from a public known processor where cryptographic accelerators are implemented. That modules are referenced under the name of "Crypto Box" in the second peripheral domain of the CPU sub-system.

This Figure has been taken from the IEEE Journal of solid-state circuits. The article is "A 90-nm CMOS Low-Power GSM/EDGE Multimedia-Enhanced Baseband Processor With 380-MHz ARM926 Core and Mixed-Signal Extensions".

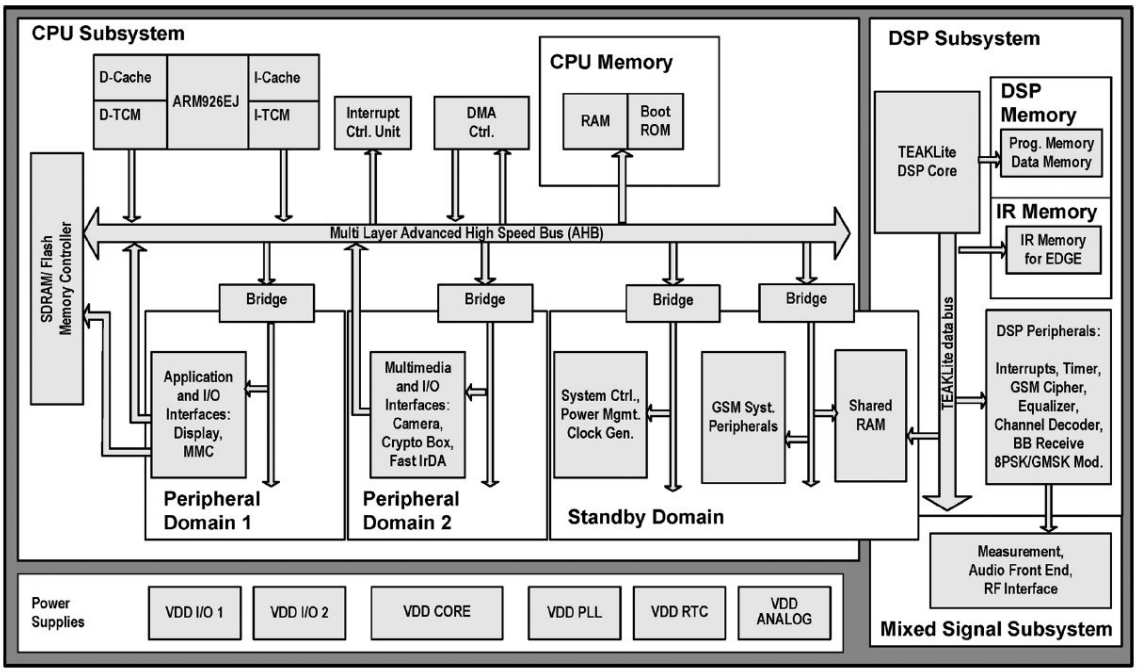


Figure 1.3 : Example of a mobile phone platform from a public known architecture

In this architecture the cryptographic accelerator (Crypto Box) is an interface connected to the Advanced High Speed Bus through an AHB Bridge. The ARM9 processor makes accesses to this hardware interface for cryptographic applications, avoiding the overload of the processors as would happen executing the cryptographic procedures entirely by software. The Standby Domain disables the Crypto Box, when not needed, for power consumption saving.

Depending from the performance requirements different architectures can also be used. If higher requirements are needed (an example may be the encryption/decryption of data packets in a router for Internet) a cryptographic coprocessor or a separate dedicated processor could be used.

1.5 Report Outline

The report is structured as follows

Chapter 2 – Cryptography : An overview of cryptography will be done focusing on the algorithms and the aspect that have been utilized during the internship project. Symmetric and Public-Key Cryptography will be explained first. Then, from this base, Authentication and MAC will be described. Finally some schemes for Key management will be illustrated.

Chapter 3 – Hardware Environment : A description of the Hardware utilized for the development of the debugging platform for the application oriented verification of Infineon's Cryptographic Encryption Unit. A description of the FPGA, in which the RTL Design has been mapped, and the ARM boards, based on the ARM9 processor, where the FPGA was connected to debug the design.

Chapter 4 – FPGA Design : In this chapter a description of the architecture that has been added around the Infineon's CEU IP will be done. Those are some CEU off-chip components that are supposed to be interfaced with it in the global architecture, and an AHB Interface to communicate with the ARM9 processor on the development boards.

Chapter 5 – Software Use Cases : In this chapter all the C software that has been written for the high level verification will be done. First an overview of all the CEU specific functions and then some SW use cases for particular applications will be described. The most important ones are: the hashing of a big area of memory, the HMAC with software and hardware flow, and the DES encryption using an AES wrapped Key.

Conclusions : The results obtained at the end of the internship.

Appendix : some information with a bit too much details that was not needed to understand the general flow of the project.

2. Cryptography

2.1 Introduction

2.1.1 Security Attacks

A useful means of classifying security attacks is in terms of passive attacks and active attacks. A passive attack attempts to learn or make use of information from the system but does not affect system resources. An active attack attempts to alter system resources or affect their operation.

Passive attacks are in the nature of eavesdropping on transmissions. The goal of the opponent is to obtain information that is being transmitted. Two types of passive attacks are release of message contents and traffic analysis. The release of message contents is the case in which we have the transfer of a file which may contain sensitive or confidential information, and we would like to prevent an opponent from learning the contents of these transmissions. Traffic analysis is when we encrypt the file before transferring it, to protect its information, but an opponent might still be able to observe the pattern of these messages. The opponent could determine the location and identity of communicating hosts and could observe the frequency and length of messages being exchanged. This information might be useful in guessing the nature of the communication that was taking place. Passive attacks are very difficult to detect because they do not involve any alteration of the data. Typically, the message traffic is sent and received in an apparently normal fashion and neither the sender nor receiver is aware that a third party has read the messages or observed the traffic pattern. However, it is feasible to prevent the success of these attacks, usually by means of encryption. Thus, the emphasis in dealing with passive attacks is on prevention rather than detection.

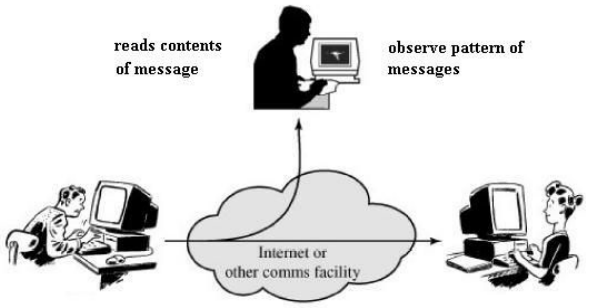


Figure 2.1 : passive attack

Active attacks involve some modification of the data stream or the creation of a false stream and can be subdivided into four categories: masquerade, replay, modification of messages, and denial of service. A masquerade takes place when one entity pretends to be a different entity. Replay involves the passive capture of a data unit and its subsequent retransmission to produce an unauthorized effect. The denial of service prevents or inhibits the normal use or management of communications facilities. Modification of messages simply means that some portion of a legitimate message is altered, or that messages are delayed or reordered, to produce an unauthorized effect. A situation in which we have an active attack is, for example, the next one: authentication sequences can be captured and replayed to

enable an authorized entity with few privileges to obtain extra privileges by impersonating an entity that has those privileges. Active attacks are difficult to prevent absolutely because of the wide variety of potential physical, software, and network vulnerabilities. Instead, the goal is to detect active attacks and to recover from any disruption or delays caused by them. In some cases detection may also contribute to prevention.

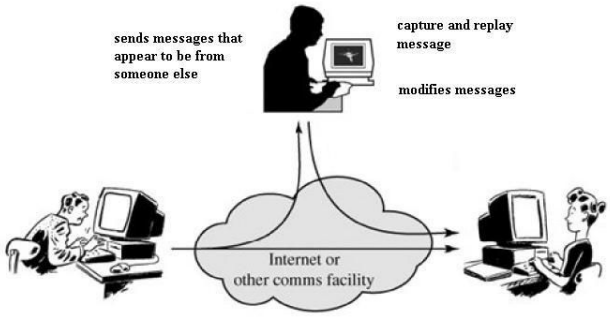


Figure 2.2 : active attack

2.1.2 Security Services

A security service is a service provided by a protocol layer, of communicating open systems, which ensures adequate security of the systems or of data transfers. Security services can be classified in the following categories: authentication, access control, data confidentiality, data integrity and nonrepudiation.

The authentication service is concerned with assuring that a communication is authentic checking that the message is from the source that it claims to be from. In the case of an ongoing interaction, at the time of connection initiation, the service assures that the two entities are authentic: each entity is the one that it claims to be. The service must also assure that the connection is not being interfered by a third party that can masquerade as one of the two legitimate parties.

In the context of network security, access control is the ability to limit and control the access to host systems and applications via communications links. To achieve this, each entity trying to gain access must first be identified, or authenticated, so that access rights can be tailored to the individual.

Confidentiality is the protection of transmitted data from passive attacks and also the protection of traffic flow from analysis. The last kind of protection requires that an attacker must not be able to observe the source and destination, frequency, length, or other characteristics of the traffic on a communication network.

A integrity service assures that messages are received as sent, with no duplication, insertion, modification, reordering, or replays. The destruction of data is also covered under this service. The integrity service relates to active attacks, so we are concerned with detection rather than prevention. If a violation of integrity is detected, then the service may simply report this violation. Alternatively there are mechanisms for automated recovery.

Non-repudiation prevents either sender or receiver from denying a transmitted message. Thus, when a message is sent, the receiver can prove that the alleged sender in fact sent the message. Similarly, when a message is received, the sender can prove that the alleged receiver in fact received the message.

An availability service is one that protects a system to ensure its availability. This service addresses the security concerns raised by denial-of-service attacks.

2.1.3 Network security

A general model for network is illustrated in the following figure.

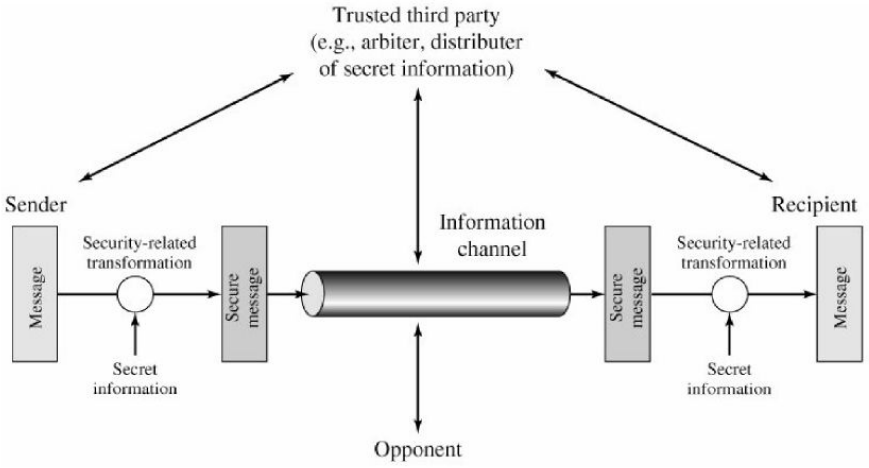


Figure 2.3 : general model for network security

A message is to be transferred from one party to another across some sort of network. The two parties, must cooperate for the exchange to take place. A logical information channel is established by defining a route through the network. Security aspects come into play when it is necessary or desirable to protect the information transmission from an opponent who may present a threat to confidentiality, authenticity, and so on. All the techniques for providing security have two components:

- A security-related transformation on the information to be sent. Examples include the encryption of the message, which scrambles the message so that it is unreadable by the opponent, and the addition of a code based on the contents of the message, which can be used to verify the identity of the sender.
- Some secret information shared by the two principals and, it is hoped, unknown to the opponent. An example is an encryption key used in conjunction with the transformation to scramble the message before transmission and unscramble it on reception.

A trusted third party may be needed to achieve secure transmission. For example, a third party may be responsible for distributing the secret information to the two principals while keeping it from any opponent. Or a third party may be needed to arbitrate disputes between the two principals concerning the authenticity of a message transmission.

This general model shows that there are four basic tasks in designing a particular security service:

- Design an algorithm for performing the security-related transformation.
- Generate the secret information to be used with the algorithm.
- Develop methods for the distribution and sharing of the secret information.
- Specify a protocol to be used by the two principals that makes use of the security algorithm and the secret information to achieve a particular security service.

.

2.2 Symmetric Algorithms

2.2.1 Symmetric Cipher Model

A general symmetric algorithm scheme is illustrated in the following figure:

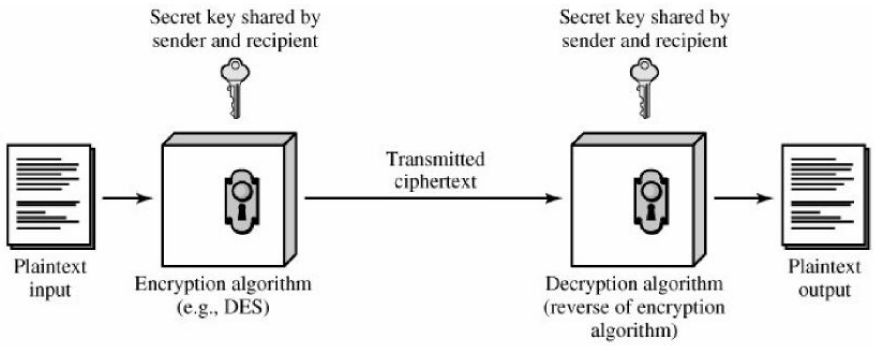


Figure 2.4 : Symmetric cipher model

There are few elements which are the basic ingredients for the symmetric algorithms, those elements are:

- Plaintext: This is the original intelligible message or data that is given to the algorithm as input.

- Encryption algorithm: The encryption algorithm performs various substitutions and transformations on the plaintext.

- Secret key: The secret key is also input to the encryption algorithm. The key is a value independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key. For symmetric algorithms the same key is used both for encryption and decryption.

- Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and, as it stands, is unintelligible.

- Decryption algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

There are two requirements for secure use of conventional encryption:

- We need a strong encryption algorithm. At a minimum, we would like the algorithm to be such that an opponent who knows the algorithm and has access to one or more ciphertexts would be unable to decipher the ciphertext or figure out the key. This requirement is usually stated in a stronger form: The opponent should be unable to decrypt ciphertext or discover the key even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.

- Sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key is readable.

We assume that it is impractical to decrypt a message on the basis of the ciphertext plus knowledge of the encryption/decryption algorithm. In other words, we do not need to keep the algorithm secret; we need to keep only the key secret. This feature of symmetric encryption is what makes it feasible for widespread use. The fact that the algorithm need not be kept secret means that manufacturers can and have developed low-cost chip implementations of data encryption algorithms. These chips are widely available and incorporated into a number of products. With the use of symmetric encryption, the principal security problem is maintaining the secrecy of the key.

2.2.2 Attacks on Encryption Systems

Typically, the objective of attacking an encryption system is to recover the key in use rather than simply to recover the plaintext of a single ciphertext. There are two general approaches to attack a conventional encryption scheme:

- Cryptanalysis: Cryptanalytic attacks rely on the nature of the algorithm plus perhaps some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used.

- Brute-force attack: The attacker tries every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained. On average, half of all possible keys must be tried to achieve success. If either type of attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised.

If either type of attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised. Another kind of attack could be classified as a :

- side channel attack : that kind of attack is based on gaining information from the physical implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms (as in cryptanalysis). Many side-channel attacks require considerable technical knowledge of the internal operation of the system on which the cryptography is implemented. An example of side channel attacks are the folowings:
    - Timing attacks : In essence, a timing attack is one in which information about the key or the plaintext is obtained by observing how long it takes a given implementation to perform decryptions on various ciphertexts. A timing attack exploits the fact that an encryption or decryption algorithm often takes slightly different amounts of time on different inputs. This is a long way from knowing the actual key, but it is an intriguing first step.
    - Architectural side-effect attacks : attacks which take advantage of side-effects of performing a computation on a particular machine architecture.
    - Power monitoring attack : attacks which make use of varying power consumption by the hardware during computation.
    - TEMPEST (or radiation monitoring) attack : attacks based on leaked electromagnetic radiation which can directly provide plaintexts and other information.
    - Acoustic cryptanalysis : attacks which exploit sound produced during a computation (rather like power analysis).

Two more definitions are worthy of note. An encryption scheme is unconditionally secure if the ciphertext generated by the scheme doesn't contain enough information to determine uniquely the corresponding plaintext, no matter how much ciphertext is available. That is, no matter how much time an opponent has, it is impossible for him or her to decrypt the ciphertext, simply because the required information is not there. With the exception of a scheme known as the one-time pad, useful primarily for low-bandwidth channels requiring very high security because of its realization difficulties, there is no encryption algorithm that is unconditionally secure. Therefore, one or both of following criteria are the goals for an encryption algorithm:

- The cost of breaking the cipher exceeds the value of the encrypted information.

- The time required to break the cipher exceeds the useful lifetime of the information.

An encryption scheme is said to be computationally secure if either of the foregoing two criteria are met.

2.2.3 Block Ciphers VS. Stream Ciphers

A stream cipher is one that encrypts a digital data stream one bit or one byte at a time. A block cipher is one in which a block of plaintext is treated as a whole and used to produce a ciphertext block

of equal length. In general block ciphers seem applicable to a broader range of applications than stream ciphers. The vast majority of network-based symmetric cryptographic applications make use of block ciphers.

Block Ciphers :

Many block ciphers have a Feistel structure. Such a structure consists of a number of identical rounds of processing. In each round, a substitution is performed on one half of the data being processed, followed by a permutation that interchanges the two halves. The original key is expanded so that a different key is used for each round.

The Data Encryption Standard (DES) has been the most widely used encryption algorithm until recently. It exhibits the classic Feistel structure. DES uses a 64-bit block and a 56-bit key. Two important methods of cryptanalysis are differential cryptanalysis and linear cryptanalysis. DES has been shown to be highly resistant to these two types of attack.

Two methods for frustating statistical cryptanalysis are diffusion and confusion. Diffusion is based on affecting the values of many cyphertext digits by each plaintext digit; generally this is equivalent to having each cyphertext digit being affected by many plaintext digits. The mechanism of diffusion seeks to make the statistical relationship between the plaintext and ciphertext as complex as possible in order to avoid the deduction of the key. On the other hand, confusion seeks to make the relationship between the statistics of the ciphertext and the value of the encryption key as complex as possible, again to avoid the deduction of the key. Thus, even if the attacker can get some handle on the statistics of the ciphertext, the way in which the key was used to produce that ciphertext is so complex as to make it difficult to deduce the key. This is achieved by the use of a complex substitution algorithm.

Since DES adoption as a federal standards, that symmetric algorithm doesn't provide anymore a high level of security. Today DES has been abandoned: because of its key size, DES is vulnerable to a brute-force attack. This problem can be solved with multiple encryption, which is a technique in which an encryption algorithm is used multiple times. In the first instance, plaintext is converted to ciphertext using the encryption algorithm. This ciphertext is then used as input and the algorithm is applied again. This process may be repeated through any number of stages.

The overall scheme for DES encryption is illustrated in Figure 2.5. As with any encryption scheme, there are two inputs to the encryption function: the plaintext to be encrypted and the key. In this case, the plaintext must be 64 bits in length and the key is 56 bits in length. Looking at the left-hand side of Figure 2.5, we can see that the processing of the plaintext proceeds in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that rearranges the bits to produce the permuted input. This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last (sixteenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation (IP-1) that is the inverse of the initial permutation function, to produce the 64-bit ciphertext. With the exception of the initial and final permutations, DES has the exact structure of a Feistel cipher.

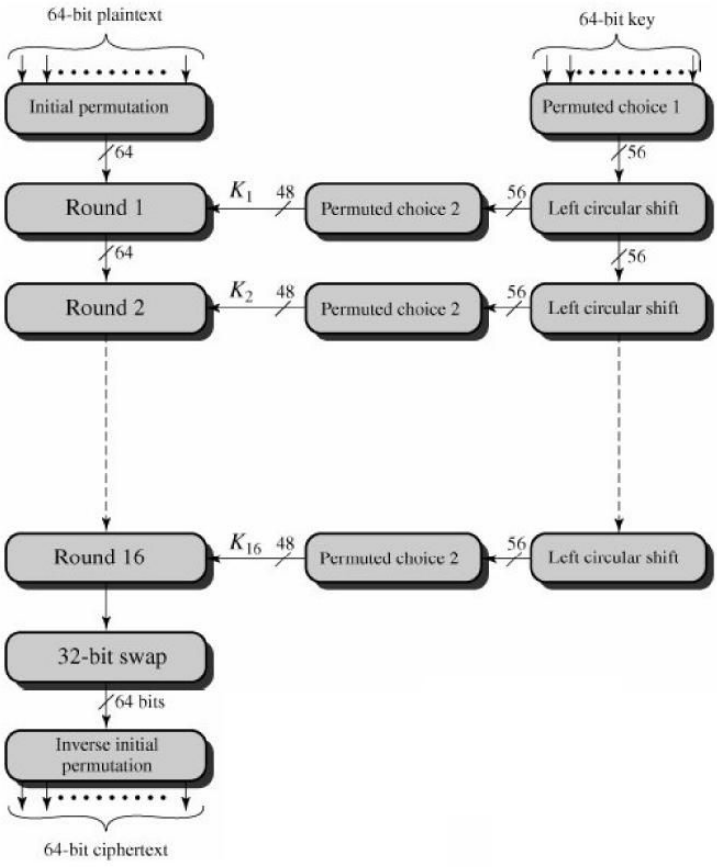


Figure 2.5 : DES Encryption scheme

Triple DES makes use of three stages of the DES algorithm, using a total of two or three distinct keys. With a triple encryption method that uses only two keys the function follows an encrypt-decrypt-encrypt (EDE) sequence C = E(K1, D(K2, E(K1, P))). 3DES with two keys is a relatively popular alternative to DES and has been adopted for use in the key management standards. Three-key 3DES has an effective key length of 112 (56x2) bits and is defined as follows: C = E(K3, D(K2, E(K1, P))). Using 3 keys we have the effective strength of 2 keys because discovering them the third one is computable. Decryption requires that the keys be applied in reverse order.

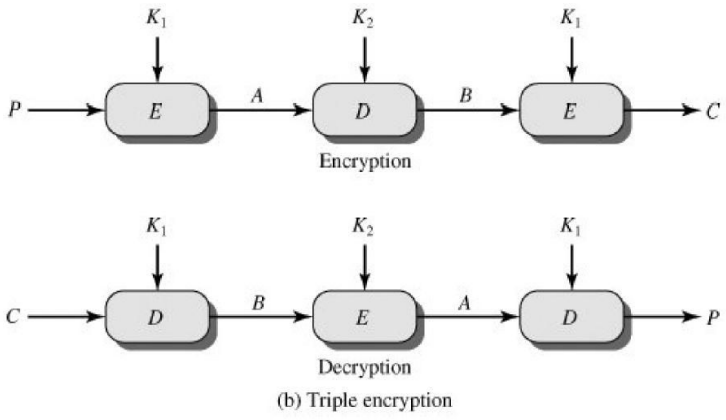


Figure 2.6 : 3DES Encryption scheme

Another approach on single DES vulnerability to brute-force attack is to design a completely new algorithm, of which AES is a an example. AES uses a 128-bit block size and a key size of 128, 192, or 256 bits. AES does not use a Feistel structure. Instead, each full round consists of four separate functions: byte substitution, permutation, arithmetic operations, and XOR with a key. The following figure shows the overall structure of AES, accordingly to the Rijndael proposal :

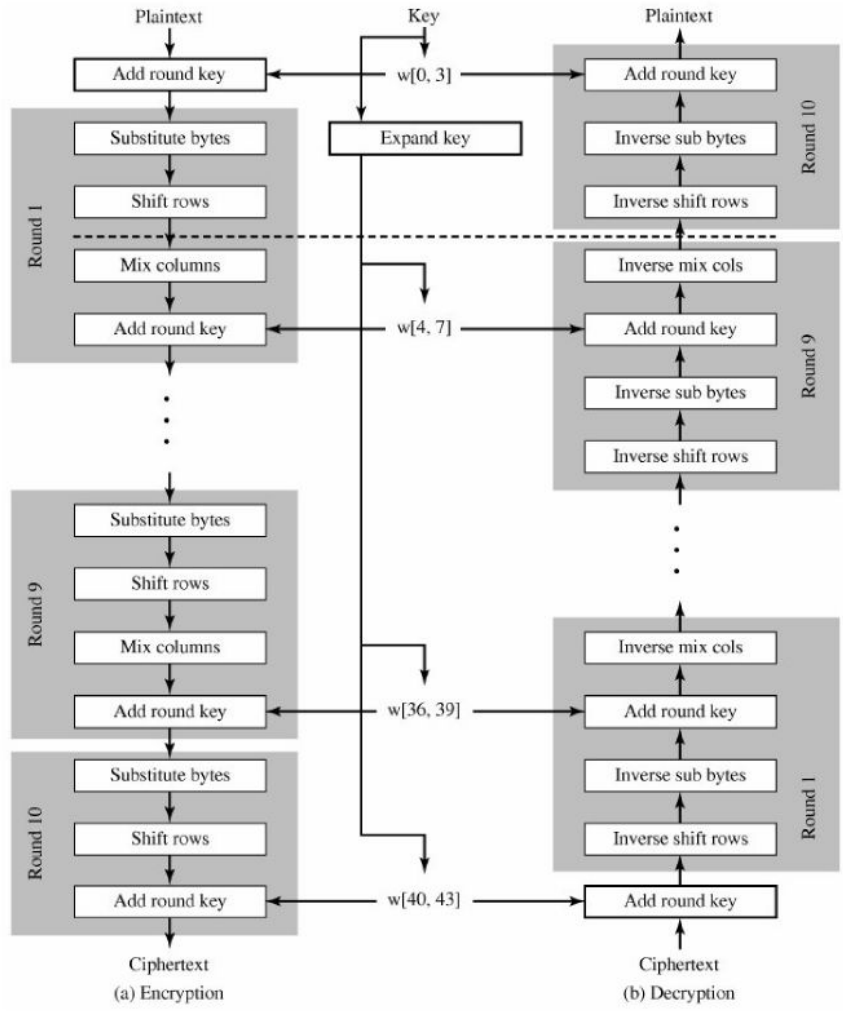


Figure 2.7 : AES Encryption scheme

Stream ciphers :

A typical stream cipher encrypts plaintext one byte at a time, although a stream cipher may be designed to operate on one bit at a time or on units larger than a byte at a time. Figure 2.8 represents the diagram of stream cipher structure. In this structure a key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random. The output of the generator, called a keystream, is combined one byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation.

A list of important design considerations for a stream cipher follows :

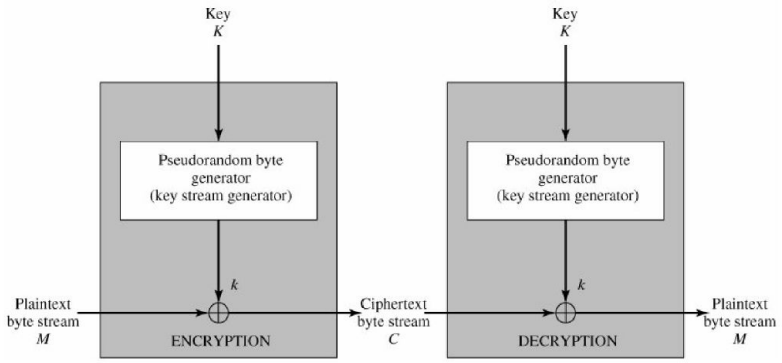


Figure 2.8 : Stream cipher scheme

- The encryption sequence should have a large period. A pseudorandom number generator uses a function that produces a deterministic stream of bits that eventually repeats. The longer the period of repeat the more difficult it will be to do cryptanalysis.

- The keystream should approximate the properties of a true random number stream as close as possible. For example, there should be an approximately equal number of 1s and 0s.

- To guard against brute-force attacks, the key needs to be sufficiently long

With a properly designed pseudorandom number generator, a stream cipher can be as secure as a block cipher of comparable key length. The primary advantage of a stream cipher is that stream ciphers are almost always faster and use far less code than do block ciphers. The advantage of a block cipher is the reuse of keys.

For applications that require encryption/decryption of a stream of data, such as over a data communications channel or a browser/Web link, a stream cipher might be the better alternative. For applications that deal with blocks of data, such as file transfer, e-mail, and database, block ciphers may be more appropriate. However, either type of cipher can be used in virtually any application.

2.3 Asymmetric Algorithms

Asymmetric encryption is a form of cryptosystem in which encryption and decryption are performed using different keys: one public key and one private key. It is also known as public-key encryption. Asymmetric encryption transforms plaintext into ciphertext using a one of two keys and an encryption algorithm. Using the paired key and a decryption algorithm, the plaintext is recovered from the ciphertext. That kind of algorithm can be used for confidentiality, authentication, or both. The most widely used public-key cryptosystem is RSA. The difficulty of attacking RSA is based on the difficulty of finding the prime factors of a composite number. Public-key cryptography provides a radical departure from all that has gone before. For one thing, public-key algorithms are based on mathematical functions rather than on substitution and permutation.

2.3.1 Public-Key Cipher Model

Asymmetric algorithms rely on one key for encryption and a different but related key for decryption. These algorithms have the following important characteristic: it is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key. In addition, some algorithms, such as RSA, also exhibit the following characteristic: either of the two related keys can be used for encryption, with the other used for decryption.

A general scheme is illustrated in Figure 2.9, from which we can see that a public-key encryption scheme has six fundamental ingredients:

- Plaintext: This is the readable message or data that is fed into the algorithm as input.
- Encryption algorithm: The encryption algorithm performs various transformations on the plaintext.
- Public key and private key: This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.
- Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
- Decryption algorithm: This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

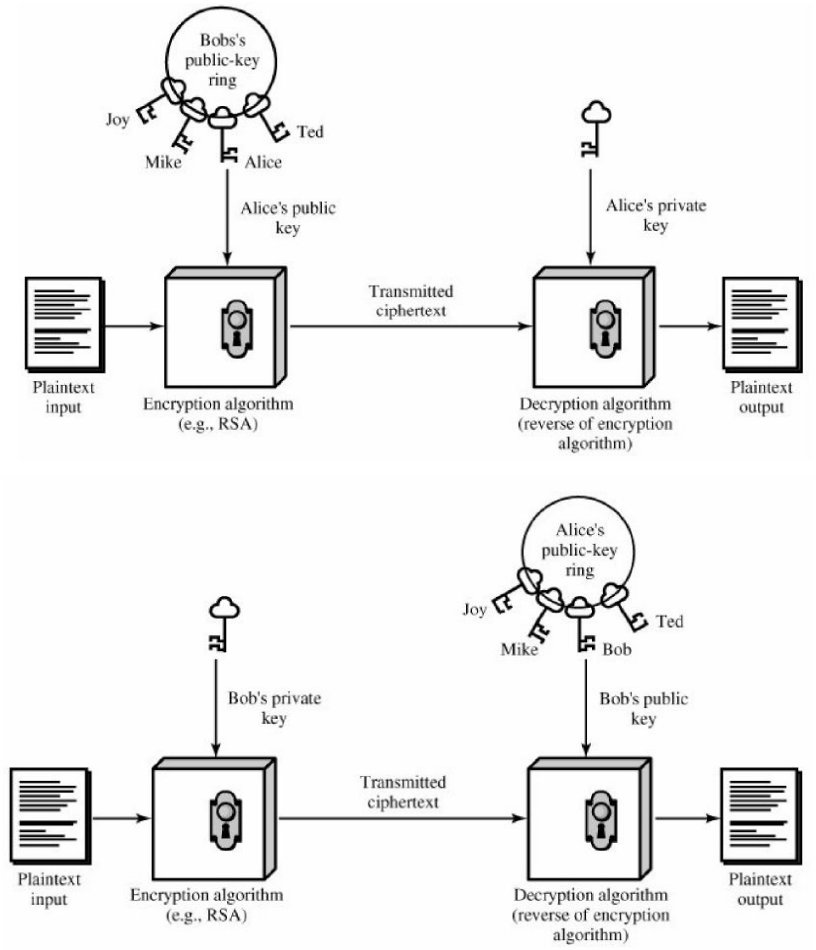


Figure 2.9 : Public key cipher model

The essential steps are the following:

- Each user generates a pair of keys to be used for the encryption and decryption of messages.
- Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. Each user maintains a collection of public keys obtained from others.
- If Bob wishes to send a confidential message to Alice, Bob encrypts the message using Alice's public key.
- When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows Alice's private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed. As long as a user's private key remains

protected and secret, incoming communication is secure. At any time, a system can change its private key and publish the companion public key to replace its old public key.

2.3.2 Applications for Public-Key Cryptosystem

We mentioned earlier that either of the two related keys can be used for encryption, with the other being used for decryption. The scheme illustrated in Figure 2.10 provides confidentiality, Figure 2.11 shows the use of public-key encryption to provide authentication.

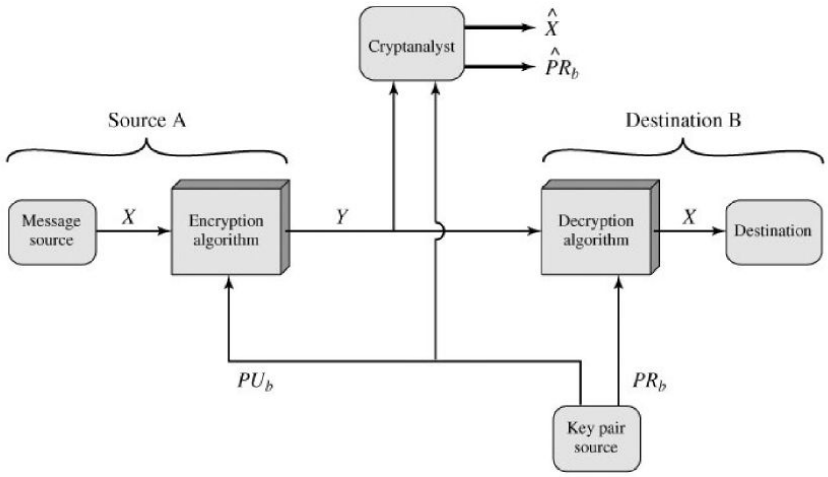


Figure 2.10 : Public key cryptosystem: Confidentiality

Let's start from the confidentiality application. There is some source A that produces a message in plaintext: X. The message is intended for destination B. B generates a related pair of keys: a public key, PUb, and a private key, PRb. PRb is known only to B, whereas PUb is publicly available and therefore accessible by A. With the message X and the encryption key PUb as input, A forms the ciphertext Y: Y = E(PUb, X). The intended receiver, in possession of the matching private key, is able to invert the transformation: X = D(PRb, Y). If the receiver B keeps his private Key secret he is the only one able to decrypt Y and recover the plaintext X, even if an adversary has knowledge of the encryption (E) and decryption (D) algorithms. That is the reason this scheme is used to provide confidentiality.

The case of authentication is illustrated in Figure 2.11. A prepares a message to B and encrypts it using A's private key before transmitting it. B can decrypt the message using A's public key. Because the message was encrypted using A's private key, only A could have prepared the message. Therefore, the entire encrypted message serves as a digital signature. In addition, it is impossible to alter the message without access to A's private key, so the message is authenticated both in terms of source and in terms of data integrity.

In the scheme of Figure 2.10, the entire message is encrypted, which requires a great deal of storage. A more efficient way of achieving the same results is to encrypt a small block of bits that is a function of the document. Such a block, called an authenticator, must have the property that it is

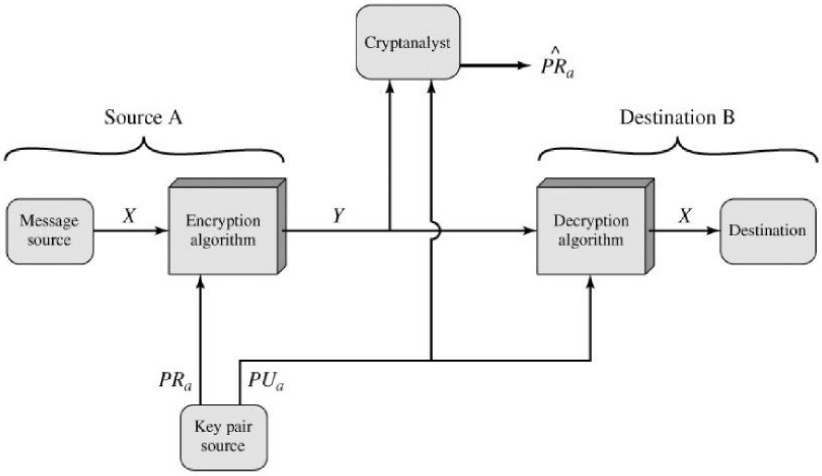


Figure 2.11 : Public key cryptosystem : Authentication

infeasible to change the document without changing the authenticator. If the authenticator is encrypted with the sender's private key, it serves as a signature that verifies the origin.

It is important to emphasize that the encryption process depicted in Figure 2.11 does not provide confidentiality. That is, the message being sent is safe from alteration but not from eavesdropping. This is obvious in the case of a signature based on a portion of the message, because the rest of the message is transmitted in the clear. Even in the case of complete encryption there is no protection of confidentiality because any observer can decrypt the message by using the sender's public key.

It is, however, possible to provide both the authentication function and confidentiality by a double use of the public-key scheme, as illustrated in the following picture

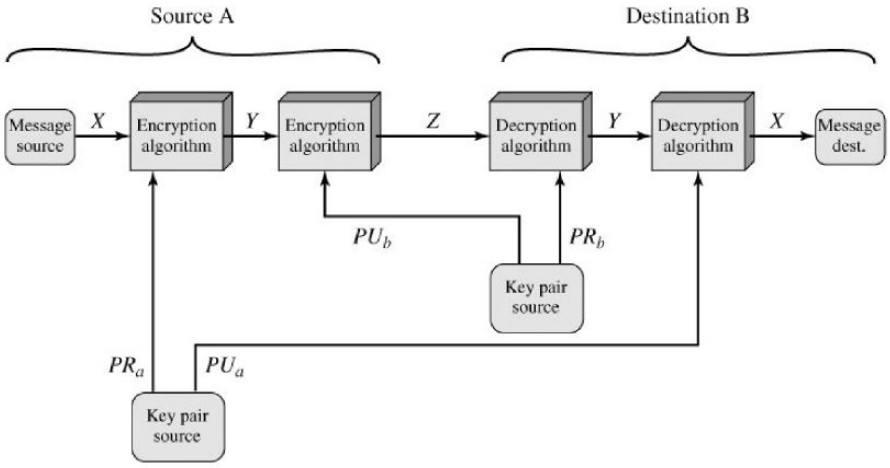


Figure 2.12 ; Public key cryptosystem : Confidentiality and Authentication

$Z = E(PUb, E(PRa, X))$, $X = D(PUa, E(PRb, Z))$

In this case, we begin as before by encrypting a message, using the sender's private key. This provides the digital signature. Next, we encrypt again, using the receiver's public key. The final ciphertext can be decrypted only by the intended receiver, who alone has the matching private key. Thus, confidentiality is provided. The recovery of the plaintext using A’s public key means that it has

been encrypted with A's private key: so we can be sure that A is the sender, thus authentication is also provided. The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.

We can resume that depending on the application, the sender uses either the sender's private key or the receiver's public key, or both, to perform some type of cryptographic function. In broad terms, we can classify the use of public-key cryptosystems into three categories:

- Encryption/decryption: The sender encrypts a message with the recipient's public key.
- Digital signature: The sender "signs" a message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.
- Key exchange: Two sides cooperate to exchange a session key. Several different approaches are possible, involving the private key(s) of one or both parties.

Those applications will be described with more details in chapters 2.5 and 2.6.

2.3.3 Requirements for Public-Key Cryptography

The Public Key cryptosystem illustrated must respect the following requirements:

1. It is computationally easy for a party B to generate a pair (public key PUb, private key PRb).
2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext: C = E(PUb, M)
3. It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message: M = D(PRb, C) = D[PRb, E(PUb, M)]
4. It is computationally infeasible for an adversary, knowing the public key, PUb, to determine the private key, PRb.
5. It is computationally infeasible for an adversary, knowing the public key, PUb, and a ciphertext, C, to recover the original message, M.

We can add a sixth requirement that, although useful, is not necessary for all public-key applications:

6. The two keys can be applied in either order: M = D[PUb, E(PRb, M)] = D[PRb, E(PUb, M)]

2.4 Hash Functions

A hash function accepts a variable-size message M as input and produces a fixed-size output, referred to as a hash code H(M). A hash code does not use a key but is a function only of the input message. The hash code is also referred to as a message digest or hash value. The hash code is a function of all the bits of the message and provides integrity: a change to any bit or bits in the message results in a completely different hash code.

2.4.1 Requirements for a Hash Function

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function h = H(x) must have the following properties :

1. H can be applied to a block of data of any size.

2. H produces a fixed-length output.

3. H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.

4. For any given value h, it is computationally infeasible to find x such that H(x) = h. This is sometimes referred to in the literature as the one-way property.

5. For any given block x, it is computationally infeasible to find y x such that H(y) = H(x). This is sometimes referred to as weak collision resistance.

6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). This is sometimes referred to as strong collision resistance.

The first three properties are requirements for the practical application of a hash function to message authentication.

The fourth property, the one-way property, states that it is easy to generate a code given a message but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value. Note that even if a secret key is not an ingredient of the hash functions, it is a popular technique to append a secret value to the message before hashing.

The fifth property guarantees that an alternative message hashing to the same value as a given message cannot be found. This prevents forgery when an encrypted hash code is used.

The sixth property refers to how resistant the hash function is to attacks.

Applications which are based on those properties will be described in the following chapter.

2.4.2 Secure Hash Algorithm (SHA)

The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. The actual standards document is entitled Secure Hash Standard. SHA is based on the hash function MD4. SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512. Those different versions are compared in the table of the following figure. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1.

	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	$<2^{64}$	$<2^{64}$	$<2^{128}$	$<2^{128}$
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256
Notes: 1. All sizes are measured in bits. 2. Security refers to the fact that a birthday attack on a message digest of size *n* produces a collision with a workfactor of approximately $2^{n/2}$				

Figure 2.13 : SHA algorithms table

SHA-512:

The scheme of the SHA-512 algorithm is now being described, considering that, as already said, the other versions have the same structure. The SHA-512 algorithm takes as input a message with a maximum length of less than 2^128 bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks. Figure 2.14 describe the overall processing of a message to produce a digest. The processing consists of the following steps:

1. Append padding bits. The message is padded so that its length is congruent to 896 modulo 1024. Padding is always added, even if the message is already of the desired length. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

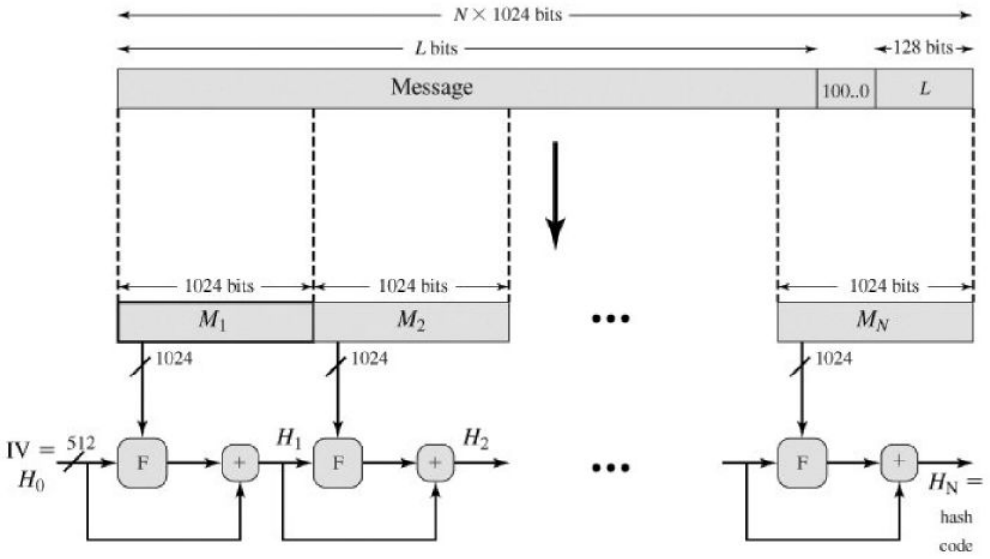


Figure 2.14 : SHA Algorithm scheme

2. Append length. A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer and contains the length of the original message (before the padding). The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. In Figure 12.1, the expanded message is represented as the sequence of 1024-bit blocks *M*1, *M*2,..., *M*N, so that the total length of the expanded message is *N* x 1024 bits.

3. Initialize hash buffer. A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are opportunely initialized.

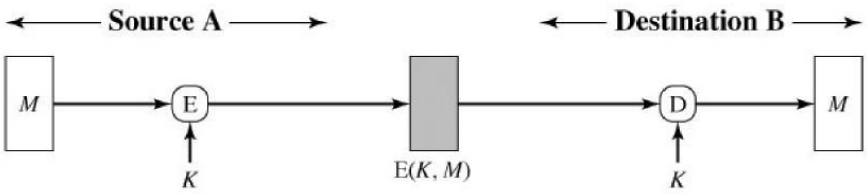
4. Process message in 1024-bit blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 2.14. IV is the initialization value. H_i is the intermediate digest.

5. Output. After all *N* 1024-bit blocks have been processed, the output from the *N*th stage is the 512-bit message digest (H_n in Figure 2.14).

2.5 Authentication

Message authentication is a mechanism or service used to verify if the sender of the message is really who he claims to be. Symmetric encryption provides authentication among those who share the secret key. Encryption of a message by a sender's private key also provides a form of authentication. The two most common cryptographic techniques for message authentication are a message authentication code (MAC) and an hash function. A MAC is an algorithm that requires the use of a secret key. A MAC takes a variable-length message and a secret key as input and produces an authentication code. A recipient in possession of the secret key can generate an authentication code to verify the integrity of the message. A hash function maps a variable-length message into a fixed length hash value, or message digest. For message authentication, a secure hash function must be combined in some fashion with a secret key.

2.5.1 Authentication with Symmetric Encryption



Symmetric encryption: confidentiality and authentication

Figure 2.15

Consider the straightforward use of symmetric encryption (Figure 2.15). A message M transmitted from source A to destination B is encrypted using a secret key K shared by A and B. If no other party knows the key, then confidentiality is provided: No other party can recover the plaintext of the message. In addition, we may say that B is assured that the message was generated by A. The message must have come from A because A is the only other party that possesses K and therefore the only other party with the information necessary to construct ciphertext that can be decrypted with K. Furthermore, if M is recovered, B knows that none of the bits of M have been altered, because an opponent that does not know K would not know how to alter bits in the ciphertext to produce desired changes in the plaintext. So we may say that symmetric encryption provides authentication as well as confidentiality. However, this flat statement needs to be better qualified.

Let’s examine in more details what is happening at B location. Given a decryption function D and a secret key K, the destination will accept any input X and produce output Y = D(K, X). If the message M could be any pattern (a stream of bits with no correlations) there is no way that B could deduce if A was the sender of the message, cause anyone could send a random message to B just to cause confusion. If instead the message is, for example, English text, B can deduce after decryption that, if Y = D(K, X) results in intelligible text, the sender was surely A. It would be infeasible for a

third party, without the knowledge of the secret key, to send a message that would give a intelligible English text after decryption.

One solution to this problem is to force the plaintext to have some structure that is easily recognized but that cannot be replicated without recourse to the encryption function. We could, for example, append an error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption, as illustrated in the following picture.

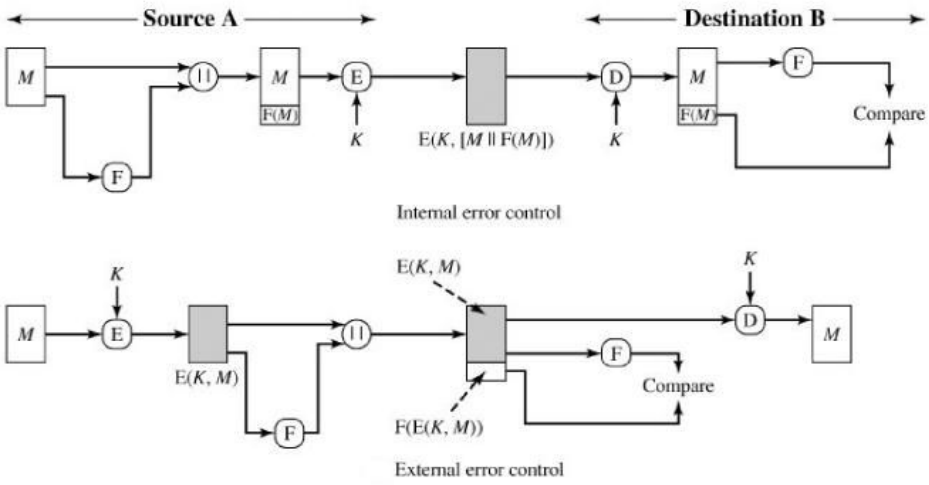
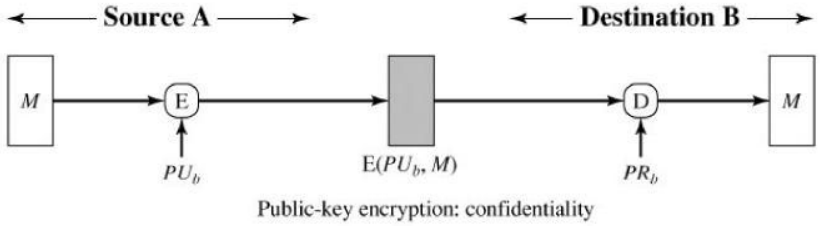


Figure 2.16 : Symmetric encryption: internal and external error control

A prepares a plaintext message M and then provides this as input to a function F that produces an FCS. The FCS is appended to M and the entire block is then encrypted. At the destination, B decrypts the incoming block and treats the results as a message with an appended FCS. B applies the same function F to attempt to reproduce the FCS. If the calculated FCS is equal to the incoming FCS, then the message is considered authentic. It is unlikely that any random sequence of bits would exhibit the desired relationship.

With internal error control, authentication is provided because an opponent would have difficulty generating ciphertext that, when decrypted, would have valid error control bits. If instead the FCS is the outer code (external error control), an opponent can construct messages with valid error-control codes. Although the opponent cannot know what the decrypted plaintext will be, he can still hope to create confusion and disrupt operations. The internal error control algorithm is to be preferred to the external one.

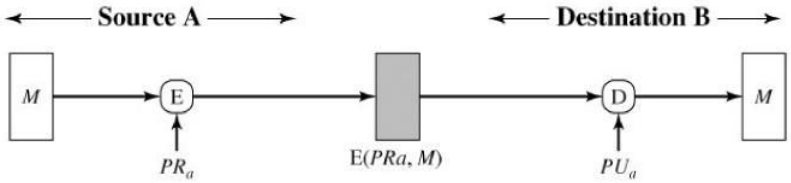
2.5.2 Authentication with Public-Key Encryption



Public-key encryption: confidentiality

Figure 2.17

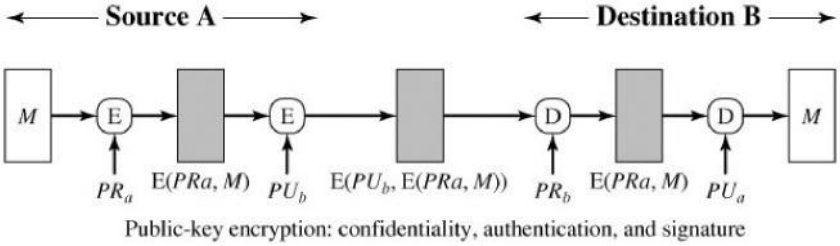
The public-key encryption scheme illustrated in Figure 2.17 provides confidentiality but not authentication. The source (A) uses the public key PUb of the destination (B) to encrypt M. Because only B has the corresponding private key PRb, only B can decrypt the message. This scheme provides no authentication because any opponent could also use B's public key to encrypt a message, claiming to be A.



Public-key encryption: authentication and signature

Figure 2.18

To provide authentication, A uses its private key to encrypt the message, and B uses A's public key to decrypt E (Figure 2.18). This provides authentication using the same type of reasoning as in the symmetric encryption case: the message must have come from A because A is the only party that possesses PRa and therefore the only party with the information necessary to construct ciphertext that can be decrypted with PUa. Again, the same reasoning as before applies: There must be some internal structure to the plaintext so that the receiver can distinguish between well-formed plaintext and random bits. Assuming there is such structure, then the scheme provides authentication. It also provides what is known as digital signature. Only A could have constructed the ciphertext because only A possesses PRa. Not even B, the recipient, could have constructed the ciphertext. Therefore, if B is in possession of the ciphertext, B has the means to prove that the message must have come from A. In effect, A has "signed" the message by using its private key to encrypt. Note that this scheme does not provide confidentiality. Anyone in possession of A's public key can decrypt the ciphertext.



Public-key encryption: confidentiality, authentication, and signature

Figure 2.19

To provide both confidentiality and authentication, A can encrypt M first using its private key, which provides the digital signature, and then using B's public key, which provides confidentiality (Figure 2.19). The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.

2.5.3 Message Authentication Code (MAC)

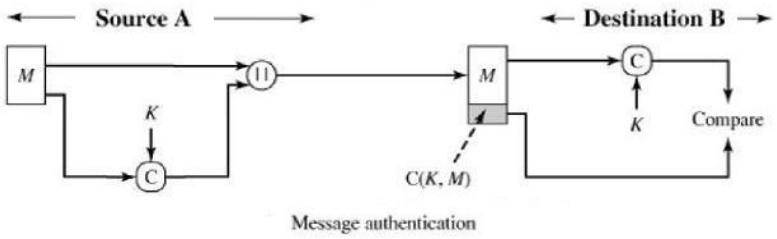


Figure 2.20

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as a cryptographic checksum or MAC that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key K. When A has a message to send to B, it calculates the MAC as a function of the message and the key: MAC = C(K, M), where

M = input message

C = MAC function

K = shared secret key

MAC = message authentication code

The message plus MAC are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new MAC. The received MAC is compared to the calculated MAC (Figure 2.20). If we assume that only the receiver and the sender know the identity of the secret key, and if the received MAC matches the calculated MAC, then

- The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the MAC, then the receiver's calculation of the MAC will differ from the received MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to correspond to the alterations in the message.

- The receiver is assured that the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper MAC.

- If the message includes a sequence number (such as is used with TCP), then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

A MAC function is similar to encryption. One difference is that the MAC algorithm need not be reversible, as it must for decryption. In general, the MAC function is a many-to-one function. Let's consider the example in which we are using 100-bit messages and a 10-bit MAC. Then, there are a total of 2^{100} different messages but only 2^{10} different MACs. So, on average, each MAC value is generated by a total of $2^{100}/2^{10} = 2^{90}$ different messages It turns out that because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption.

Because symmetric encryption will provide authentication why not simply use this instead of a separate message authentication code? This is because there are various applications in which it is preferred to send the message in plaintext instead of cyphertext. For example: an alarm which is broadcast to various destinations: authentication with a MAC is cheaper and more reliable. Another example is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages.

The process illustrated in Figure 2.20 provides authentication but not confidentiality, because the message as a whole is transmitted in the clear. Confidentiality can be provided by performing message encryption either after or before the MAC algorithm, as explained by the following figure.

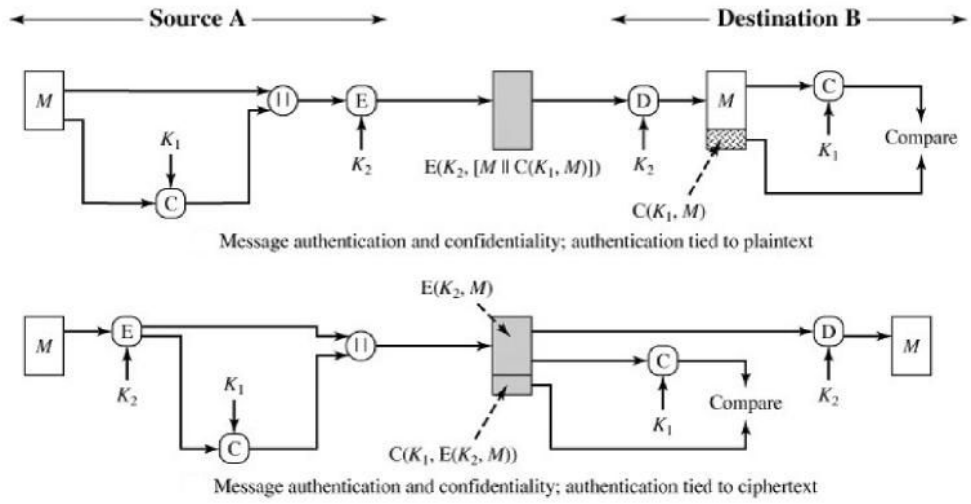


Figure 2.21

In both these cases, two separate keys are needed, each of which is shared by the sender and the receiver. In the first case, the MAC is calculated with the message as input and is then concatenated to the message. The entire block is then encrypted. In the second case, the message is encrypted first. Then the MAC is calculated using the resulting ciphertext and is concatenated to the ciphertext to form the transmitted block. Typically, it is preferable to tie the authentication directly to the plaintext.

2.5.4 Hashed Message Authentication Code (HMAC)

A variation on the message authentication code is the one-way hash function. As with the message authentication code, a hash function accepts a variable-size message M as input and produces a fixed-size output, referred to as a hash code H(M). Unlike a MAC, a hash code does not use a key but is a function only of the input message. The hash code is also referred to as a message digest or hash value. The hash code is a function of all the bits of the message and provides an error-detection capability: A change to any bit or bits in the message results in a change to the hash code.

The following Figures explain a variety of ways in which a hash code can be used to provide message authentication:

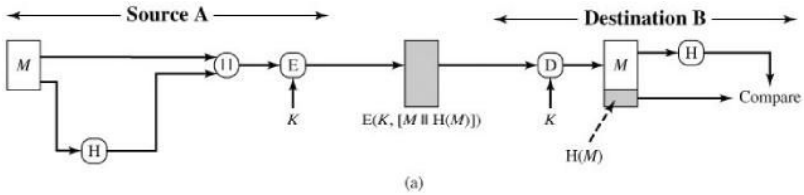


Figure 2.22 (a) : HMAC with symmetric encryption 1

a. The message plus concatenated hash code is encrypted using symmetric encryption. This is identical in structure to the internal error control strategy shown in chapter 2.5.1. The same line of reasoning applies: Because only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication. Because encryption is applied to the entire message plus hash code, confidentiality is also provided.

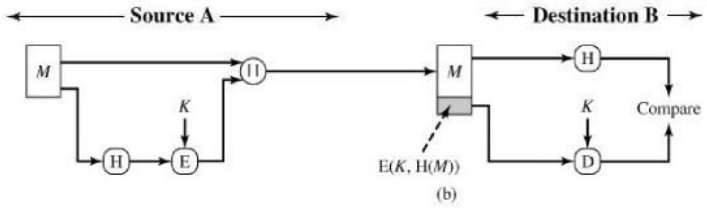


Figure 2.23 (b) : HMAC with symmetric encryption 2

b. Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality. The combination of hashing and encryption results in an overall function that is, in fact, a MAC. That is, E(K, H(M)) is a function of a variable-length message M and a secret key K, and it produces a fixed-size output that is secure against an opponent who does not know the secret key.

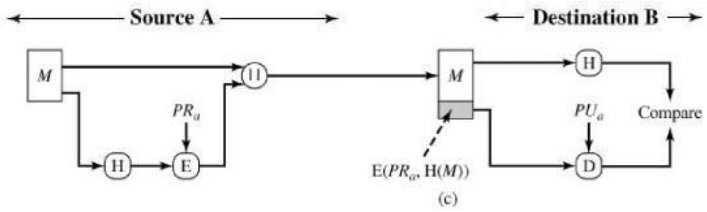


Figure 2.24 (c) : HMAC with public key encryption: digital signature scheme

c. Only the hash code is encrypted, using public-key encryption and using the sender's private key. As with (b), this provides authentication. It also provides a digital signature, because only the sender could have produced the encrypted hash code. In fact, this is the essence of the digital signature technique. This technique is the one used by INFINEON for Secure Boot. For this scenario I implemented some software for the hashing of a big block of data with context switch support. This application will be described with details in the following chapters.

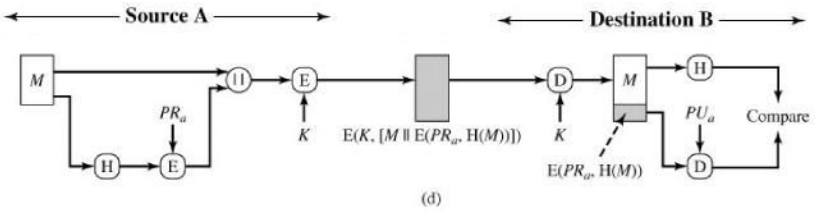


Figure 2.25 (d) : HMAC with Public key encryption : authentication and confidentiality

d. If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.

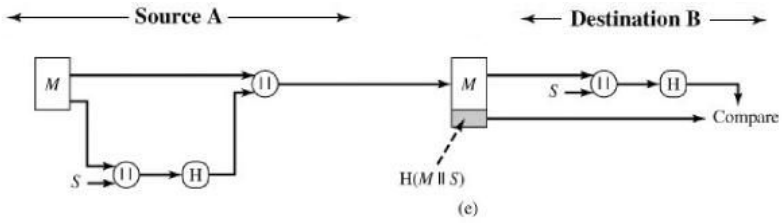


Figure 2.26 (e) : HMAC with secret value shared, authentication

e. It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value S. A computes the hash value over the concatenation of M and S and appends the resulting hash value to M. Because B possesses S, it can recompute the hash value for a verification. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.

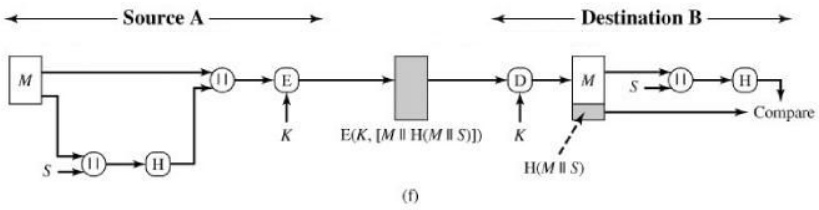


Figure 2.27 (f) : HMAC with secret value shared, authentication and confidentiality

f. Confidentiality can be added to the approach of (e) by encrypting the entire message plus the hash code.

When confidentiality is not required, methods (b) and (c) have an advantage over those that encrypt the entire message in that less computation is required.

2.6 Key Management

2.6.1 Key Distribution for Symmetric Encryption

For symmetric encryption to work, the two parties to an exchange must share the same key, and that key must be protected from access by others. Furthermore, frequent key changes are usually desirable to limit the amount of data compromised if an attacker learns the key. Therefore, the strength of any cryptographic system rests with the key distribution technique, a term that refers to the means of delivering a key to two parties who wish to exchange data, without allowing others to see the key.

For end-to-end encryption a key distribution center is responsible for distributing keys to pairs of users as needed. Each user must share a unique key with the key distribution center for purposes of key distribution. The use of a key distribution center is based on the use of a hierarchy of keys. At a minimum, two levels of keys are used. Communication between end systems is encrypted using a temporary key, often referred to as a session key. Typically, the session key is used for the duration of a logical connection, such as a frame relay connection or transport connection, and then discarded. Each session key is obtained from the key distribution center over the same networking facilities used for end-user communication. Accordingly, session keys are transmitted in encrypted form, using a master key that is shared by the key distribution center and an end system or user. For each end system or user, there is a unique master key that it shares with the key distribution center. Of course, these master keys must be distributed in some fashion.

The key distribution concept can be deployed in a number of ways. A typical scenario for key distribution is illustrated in Figure 2.28. The scenario assumes that each user shares a unique master key with the key distribution center (KDC).

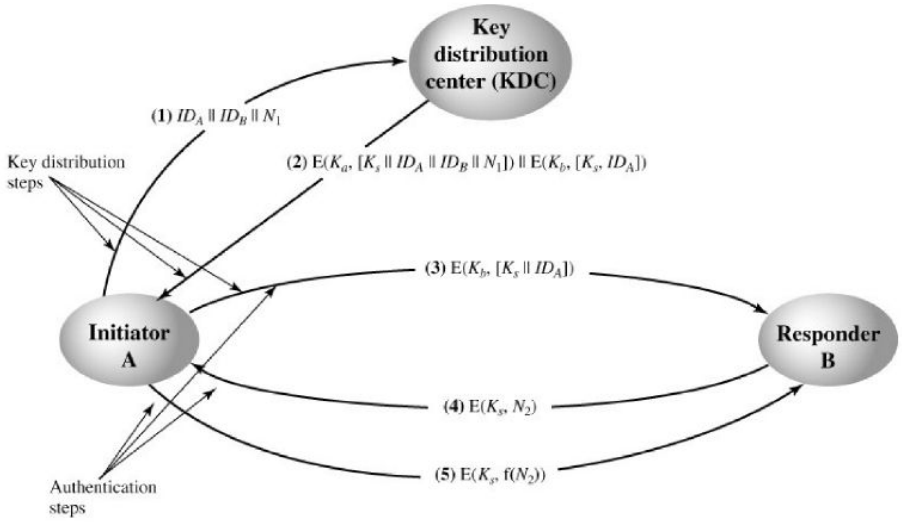


Figure 2.28 : Key distribution with symmetric encryption

Let us assume that user A wishes to establish a logical connection with B and requires a one-time session key to protect the data transmitted over the connection. A has a master key, Ka, known only to itself and the KDC; similarly, B shares the master key Kb with the KDC. The following steps occur:

1. A issues a request to the KDC for a session key to protect a logical connection to B. The message includes the identity of A and B and a unique identifier, N1, for this transaction, which we refer to as a nonce. The nonce may be a timestamp, a counter, or a random number; the minimum requirement is that it differs with each request. A random number is a good choice for a nonce.

2. The KDC responds with a message encrypted using Ka. So, A is the only one who can successfully read the message, and A knows that it originated at the KDC. The message includes two items intended for A:
    - The one-time session key, Ks, to be used for the session
    - The original request message, including the nonce, to enable A to match this response with the appropriate request

    Thus, A can verify that its original request was not altered before reception by the KDC and, because of the nonce, that this is not a replay of some previous request. In addition, the message includes two items intended for B:
    - The one-time session key, Ks to be used for the session
    - An identifier of A (for example its network address), IDA

    These last two items are encrypted with Kb (the master key that the KDC shares with B). They are to be sent to B to establish the connection and prove A's identity.

3. A stores the session key for use in the upcoming session and forwards to B the information that originated at the KDC for B, namely, E(Kb, [Ks || IDA]). Because this information is encrypted with Kb, it is protected from eavesdropping. B now knows the session key (Ks), knows that the other party is A (from IDA), and knows that the information originated at the KDC (because it is encrypted using Kb).

At this point, a session key has been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

4. Using the newly minted session key for encryption, B sends a nonce N2, to A.

5. Also using Ks, A responds with f(N2), where f is a function that performs some transformation on N2

These steps assure B that the original message it received (step 3) was not a replay. Note that the actual key distribution involves only steps 1 through 3 but that steps 4 and 5, as well as 3, perform an authentication function.

2.6.2 Distribution of Public Keys

Public-key encryption schemes are secure only if the authenticity of the public key is assured. One of the major roles of public-key encryption has been to address the problem of key distribution. There are actually two distinct aspects to the use of public-key cryptography in this regard:

- The distribution of public keys
- The use of public-key encryption to distribute secret keys

Several techniques have been proposed for the distribution of public keys. Virtually all these proposals can be grouped into the following general schemes:

- Public announcement
- Publicly available directory
- Public-key authority
- Public-key certificates

Let's examine the public announcement first. On the face of it, the point of public-key encryption is that the public key is public. Thus, if there is some broadly accepted public-key algorithm, such as RSA, any participant can send his or her public key to any other participant or broadcast the key to the community at large. Although this approach is convenient, it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be user A and send a public key to another participant or broadcast such a public key. Until such time as user A discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for A and can use the forged keys for authentication.

A greater degree of security can be achieved by maintaining a publicly available dynamic directory of public keys. Maintenance and distribution of the public directory would have to be the responsibility of some trusted entity or organization.

An even stronger security can be achieved with a public-key authority and public-key certificates. We will examine these schemes in the two following chapters.

2.6.3 Public-Key Authority

The scenario for the public-key authority scheme is illustrated in Figure 2.29. It assumes that a central authority maintains a dynamic directory of public keys of all participants. In addition, each

participant reliably knows a public key for the authority, with only the authority knowing the corresponding private key.

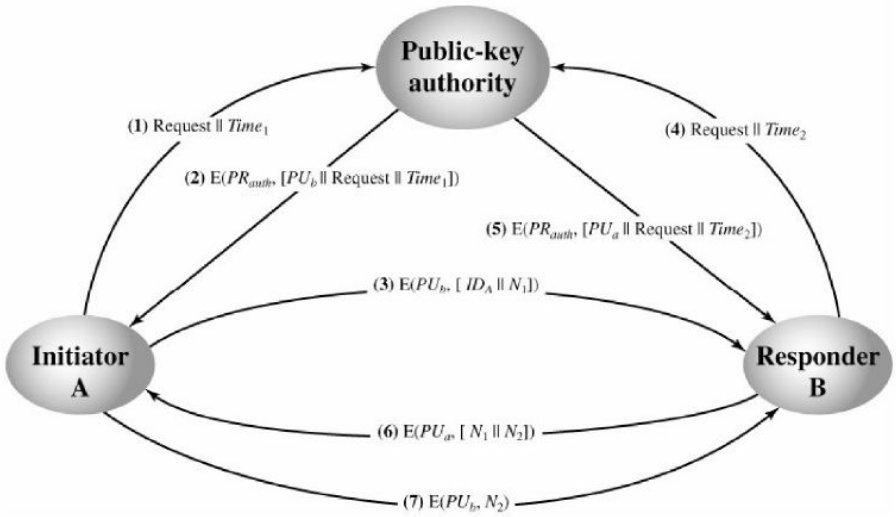


Figure 2.29 : Public key authority

The following steps are used for a secure key distribution:

1. A sends a time-stamped message to the public-key authority containing a request for the current public key of B.

2. The authority responds with a message that is encrypted using the authority's private key, PRauth Thus, A is able to decrypt the message using the authority's public key. Therefore, A is assured that the message originated with the authority. The message includes the following:
    - B's public key, PUb which A can use to encrypt messages destined for B
    - The original request, to enable A to match this response with the corresponding earlier request and to verify that the original request was not altered before reception by the authority
    - The original timestamp, so A can determine that this is not an old message from the authority containing a key other than B's current public key

3. A stores B's public key and also uses it to encrypt a message to B containing an identifier of A (IDA) and a nonce (N1), which is used to identify this transaction uniquely.

4. B retrieves A's public key from the authority in the same manner as A retrieved B's public key.

At this point, public keys have been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

5. B sends a message to A encrypted with PUa and containing A's nonce (N1) as well as a new nonce generated by B (N2). Because only B could have decrypted message (3), the presence of N1 in message (6) assures A that the correspondent is B.

6. A returns N2, encrypted using B's public key, to assure B that its correspondent is A.

A total of seven messages are required (step 4 needs 2 messages). However, the initial four messages need to be used only infrequently because both A and B can save the other's public key for future use, a technique known as caching. Periodically, a user should request fresh copies of the public keys of its correspondents to ensure currency.

2.6.4 Public-Key Certificates

The public-key authority could be somewhat of a bottleneck in the system, for a user must appeal to the authority for a public key for every other user that it wishes to contact. An alternative approach is to use certificates that can be used by participants to exchange keys without contacting a public-key authority. In essence, a certificate consists of a public key plus an identifier of the key owner, with the whole block signed by a trusted third party. Typically, the third party is a certificate authority, such as a government agency or a financial institution, that is trusted by the user community. A user can present his or her public key to the authority in a secure manner, and obtain a certificate. The user can then publish the certificate. Anyone needed this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signature. A participant can also convey its key information to another by transmitting its certificate. Other participants can verify that the certificate was created by the authority. We can place the following requirements on this scheme:

- Any participant can read a certificate to determine the name and public key of the certificate's owner.
- Any participant can verify that the certificate originated from the certificate authority and is not counterfeit.
- Only the certificate authority can create and update certificates.
- Any participant can verify the currency of the certificate.

A certificate scheme is illustrated in Figure 2.30. Each participant applies to the certificate authority, supplying a public key and requesting a certificate. Application must be in person or by some form of secure authenticated communication. For participant A, the authority provides a certificate of the form CA = E(PRauth, [T||IDA||PUa]) where PRauth is the private key used by the authority and T is a timestamp. A may then pass this certificate on to any other participant, who reads and verifies the certificate as follows: D(PUauth, CA) = D(PUauth, E(PRauth, [T||IDA||PUa])) = (T||IDA||PUa).

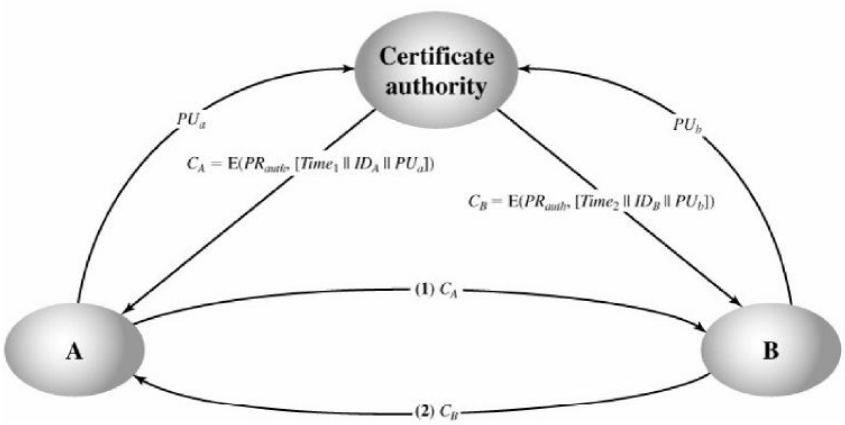


Figure 2.30 : Certificate authority

The recipient uses the authority's public key, PUauth to decrypt the certificate. Because the certificate is readable only using the authority's public key, this verifies that the certificate came from the certificate authority. The elements IDA and PUa provide the recipient with the name and public key of the certificate's holder. The timestamp T validates the currency of the certificate.

2.6.5 Simple Secret Key Distribution using Public-Key Cryptography

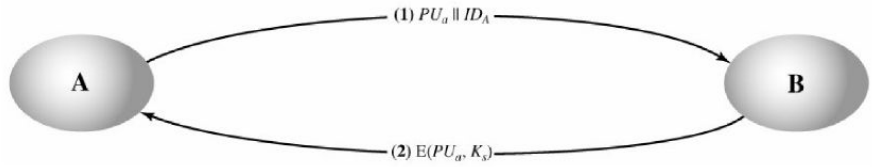


Figure 2.31 : Key distribution with public key cryptography, scheme 1

If A wishes to communicate with B, the following procedure, illustrated in Figure 2.31, can be followed:

1. A generates a public/private key pair {PUa, PRa} and transmits a message to B consisting of PUa and an identifier of A, IDA.
2. B generates a secret key, Ks, and transmits it to A, encrypted with A's public key.
3. A computes D(PRa, E(PUa, Ks)) to recover the secret key. Because only A can decrypt the message, only A and B will know the identity of Ks.
4. A discards PUa and PRa and B discards PUa.

A and B can now securely communicate using conventional encryption and the session keyK s. At the completion of the exchange, both A and B discard Ks. Despite its simplicity, this is an attractive

protocol. No keys exist before the start of the communication and none exist after the completion of communication. Thus, the risk of compromise of the keys is minimal. At the same time, the communication is secure from eavesdropping.

But the protocol is insecure against an adversary who can intercept messages and then either relay the intercepted message or substitute another message. Such an attack is known as a man-in-the-middle attack. In this case, if an adversary, E, has control of the intervening communication channel, then E can compromise the communication in the following way without being detected:

1. A generates a public/private key pair {PUa, PRa} and transmits a message intended for B consisting ofP Ua and an identifier of A, IDA.
2. E intercepts the message, creates its own public/private key pair {PUe, PRe} and transmits PUe||IDA to B.
3. B generates a secret key, Ks, and transmits E(PUe, Ks).
4. E intercepts the message, and learns Ks by computing D(PRe, E(PUe, Ks)).
5. E transmits E(PUa, Ks) to A.

The result is that both A and B know Ks and are unaware that Ks has also been revealed to E. A and B can now exchange messages using Ks. E no longer actively interferes with the communications channel but simply eavesdrops. Knowing Ks E can decrypt all messages, and both A and B are unaware of the problem. Thus, this simple protocol is only useful in an environment where the only threat is eavesdropping

2.6.6 Secret Key Distribution with Confidentiality and Authentication using Public-Key Cryptography

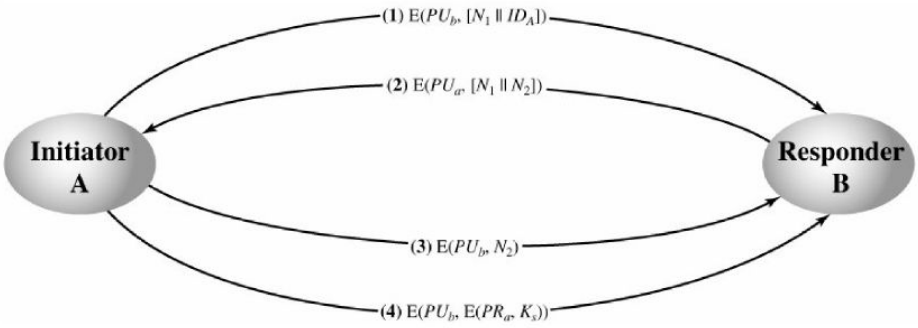


Figure 2.32 : Key distribution with public key cryptography, scheme 2

The scheme illustrated in Figure 2.32 provides protection against both active and passive attacks. We begin at a point when it is assumed that A and B have exchanged public keys by one of the schemes described in the previous chapters. Then the following steps are followed:

1. A uses B's public key to encrypt a message to B containing an identifier of A (IDA) and a nonce (N1), which is used to identify this transaction uniquely.
2. B sends a message to A encrypted with PUa and containing A's nonce (N1) as well as a new nonce generated by B (N2). Because only B could have decrypted message (1), the presence of N1 in message (2) assures A that the correspondent is B.
3. A returns N2 encrypted using B's public key, to assure B that its correspondent is A.
4. A selects a secret key Ks and sends M = E(PUb, E(PRa, Ks)) to B. Encryption of this message with B's public key ensures that only B can read it; encryption with A's private key ensures that only A could have sent it.
5. B computes D(PUa, D(PRb, M)) to recover the secret key.

The first three steps of this scheme are the same as the last three steps used in the public-key authority chapter (2.6.3). The result is that this scheme ensures both confidentiality and authentication in the exchange of a secret key.

3. Work Environment

The FPGA Design is mapped on a VIRTEX II XC2V8000 FPGA. The flip-chip Ball Grid Array Package with that device is mounted on the ARM Versatile Logic Tile XC2V4000+. The Logic Tile is connected to the Versatile Platform Baseboard for ARM926EJ-S. In the following chapters a short description of the work environment is done.

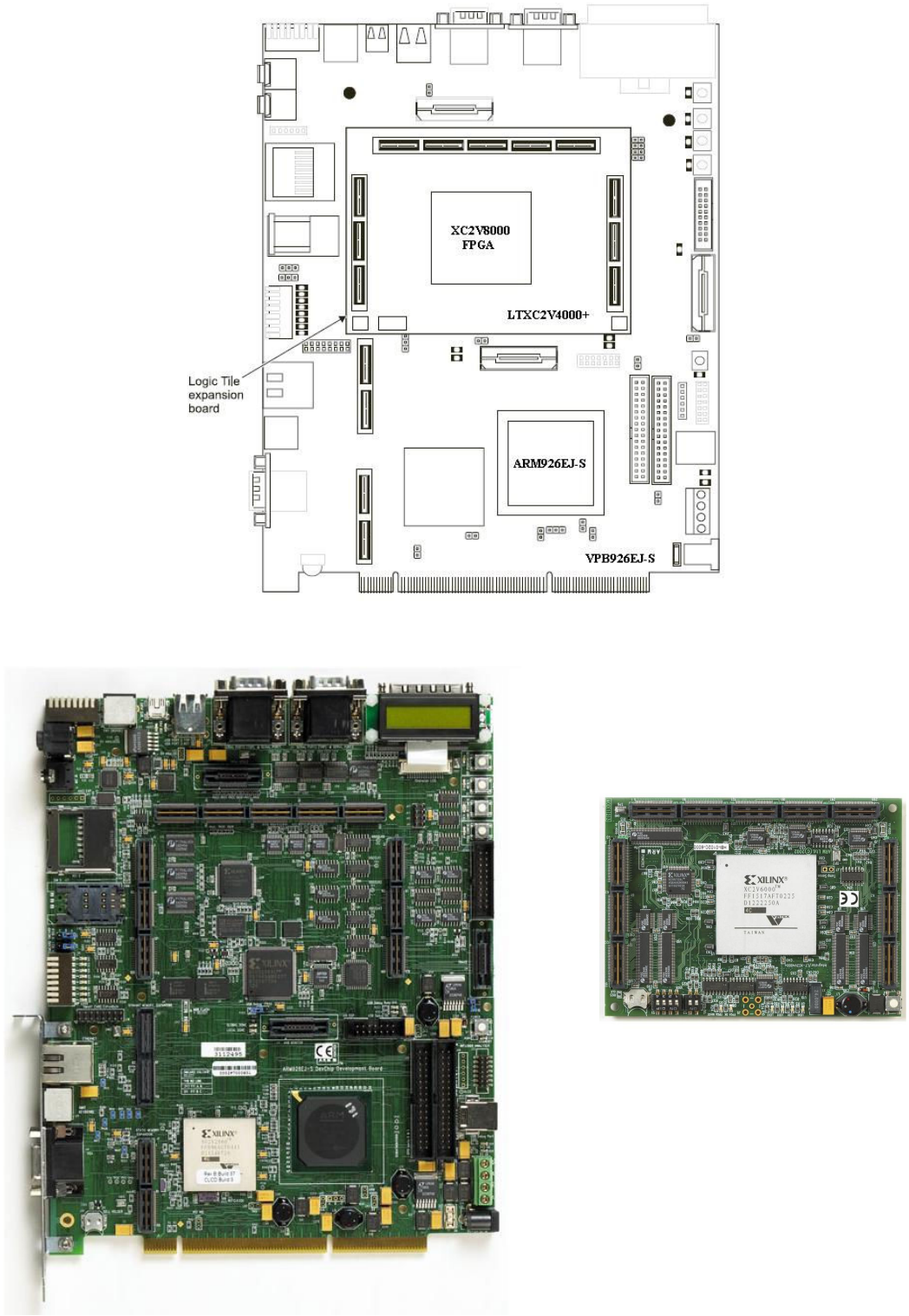


Figure 3.1 : Hardware environment

3.1 Structure of the System

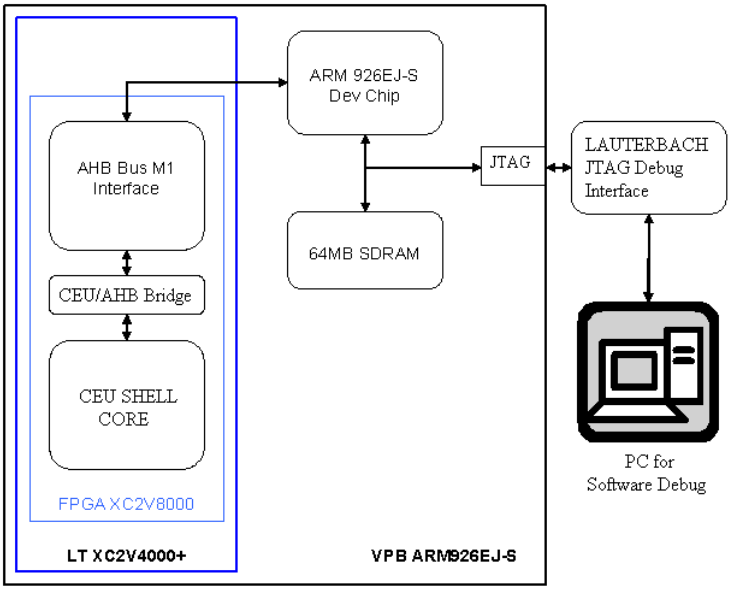


Figure 3.2 : Structure of the system

The system is implemented on two development boards:

The Versatile Platform Baseboard for ARM926EJ-S (Mother Board): contains the 32-bit ARM9 processor. This board can be programmed to run one of the crypto programs communicating with the CEU Shell mapped in the FPGA. Data can be saved/loaded to/from the SDRAM memory. The Board memory, and so all memory mapped registers and peripherals, can be controlled with Trace32 PC Debug Software.

The Logic Tile XC2V4000+ (Daughter Board): contains the Virtex-II FPGA in which the CEU SHELL is implemented together with an AHB Interface to communicate with the Mother Board Bus.

3.2 XILINX XC2V8000 FPGA

The Virtex-II family is a platform FPGA developed for high performance from low-density to high-density designs that are based on IP cores and customized modules. The leading-edge 0.15 µm / 0.12 µm CMOS 8-layer metal process and the Virtex-II architecture are optimized for high speed with low power consumption. Combining a wide variety of flexible features and a large range of densities up to 10 million system gates, the Virtex-II family enhances programmable logic design capabilities.

As shown in Figure 3.3, the programmable device is comprised of input/output blocks (IOBs) and internal configurable logic blocks (CLBs).

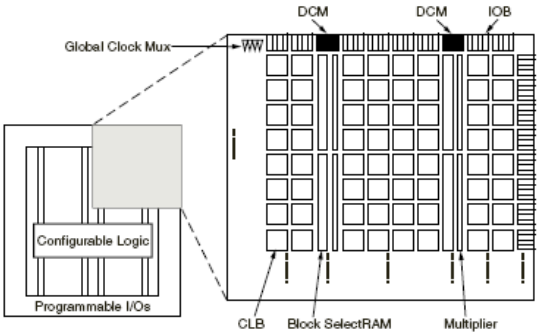


Figure 3.3 : Virtex II FPGA family architecture

The internal configurable logic includes four major elements organized in a regular array:

• Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.

• Block SelectRAM memory modules provide large 18 Kbit storage elements of dual-port RAM.

• Multiplier blocks are 18-bit x 18-bit dedicated multipliers.

• DCM (Digital Clock Manager) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, coarse- and fine-grained clock phase shifting.

A new generation of programmable routing resources called Active Interconnect Technology interconnects all of these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs. All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

The Virtex-II configurable logic blocks (CLB) are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access

the general routing matrix, as shown in Figure 3.4. A CLB element comprises 4 similar slices, with fast local feedback within the CLB. The four slices are split in two columns of two slices with two independent carry logic chains and one common shift chain. Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. As shown in Figure 3.4, each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit shift register element. The distributed SelectRAM configuration has been utilized in our FPGA Design to generate the 4 blocks of 128x32bits SRAMs in the CEU Shell.

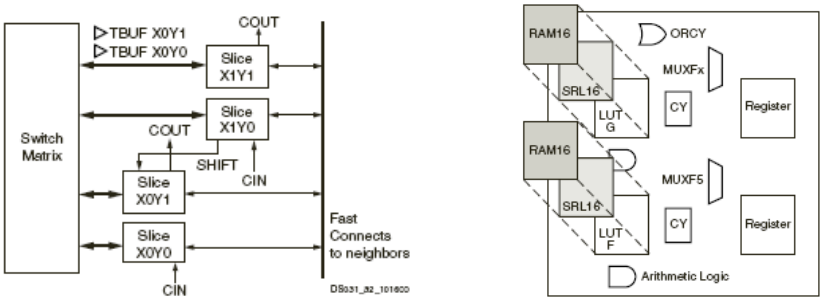


Figure 3.4 : Virtex II Configurable Logic Block (CLB)

Virtex-II devices have 16 clock input pins that can also be used as regular user I/Os. Eight clock pads are on the top edge of the device, in the middle of the array, and eight are on the bottom edge. Global clock buffers are used to distribute the clock to some or all synchronous logic elements (such as registers in CLBs and IOBs, and SelectRAM blocks. Figure 3.5 shows clock distribution in Virtex-II devices. In each quadrant, up to eight clocks are organized in clock rows. A clock row supports up to 16 CLB rows (eight up and eight down). The most common configuration option for a buffer is BUFG function, the one also used in our design.

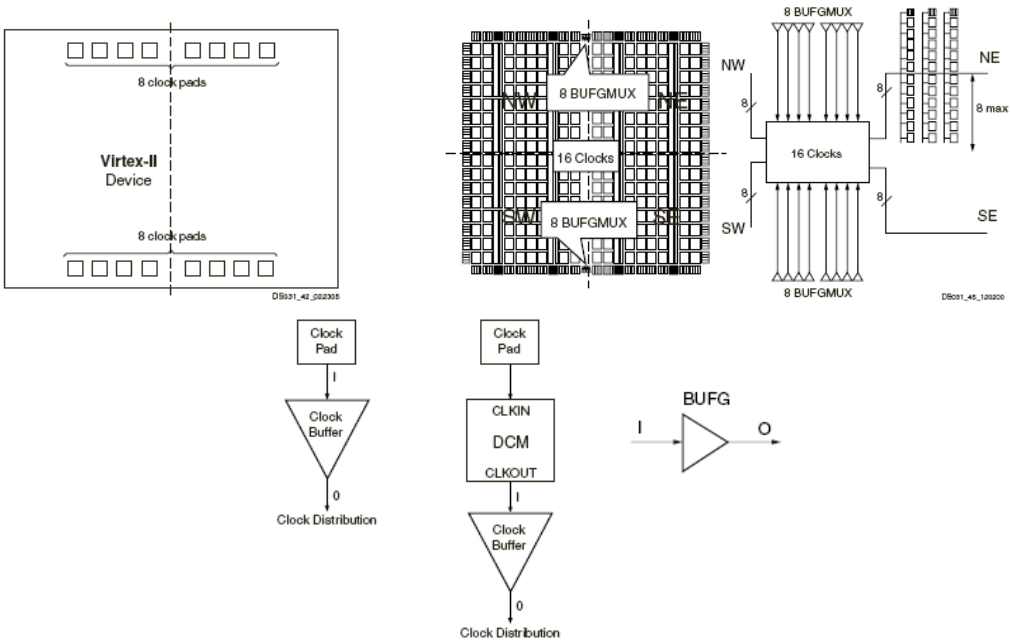


Figure 3.5 : Virtex II clock architecture

3.3 Versatile Logic Tile XC2V4000+

The LT-XC2V4000+ Logic Tile is designed as a platform for developing Advanced Microcontroller Bus Architecture (AMBA™) Advanced System Bus (ASB), Advanced High-performance Bus (AHB), Advanced Peripheral Bus (APB) peripherals, or custom logic for use with ARM cores. The functionality of the Logic Tile is defined by a configuration image loaded into the FPGA at power-up.

The Logic Tile comprises the following:

• Xilinx Virtex II FPGA

• configuration Programmable Logic Device (PLD) and flash memory for storing FPGA configurations

• Two 2MB ZBT SSRAM chips

• clock generators and reset sources

• switches

• LEDs

• battery for DES encryption keys

• connectors to other tiles.

The architecture of the Logic Tile is schematized in the following figure

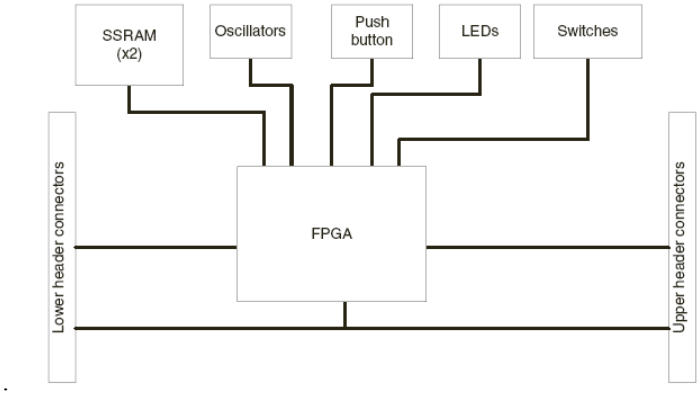


Figure 3.6 : Logic Tile architecture scheme

Three programmable (6 – 200 MHz) clocks are supplied to the I/O pins by three serially programmable MicroClock ICS307 clock generators, as shown in Figure 3.7. These are general purpose clock sources and can be used for your design. The ICS307s are supplied with a reference clock by a 24MHz crystal oscillator. The frequency of the outputs from the ICS307s are controlled by values loaded into the serial data pins. This enables them to produce a wide range of frequencies.

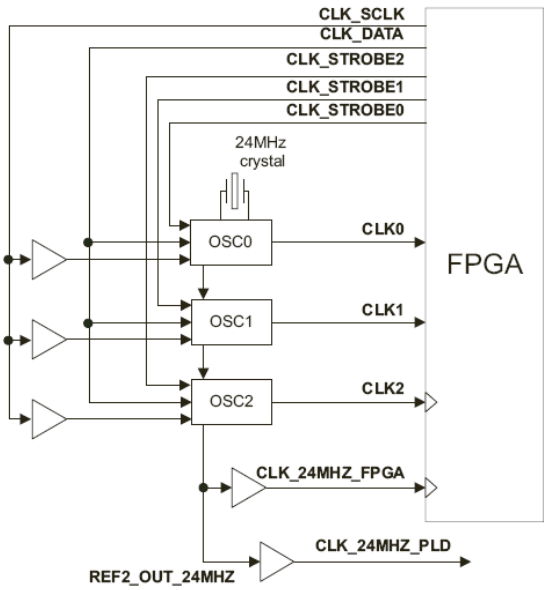


Figure 3.7 : Logic Tile clock architecture scheme

3.4 Versatile Platform Baseboard for ARM926EJ-S

The Versatile/PB926EJ-S provides a development system that can be used to develop products around the ARM926EJ-S Development Chip. In our case that system has been expanded with an ARM Real View Logic Tile containing the CEU IP. The basic system provides a good platform for developing code for the ARM7 and ARM9 series of processors. The ARM926EJ-S Development Chip is much faster than a software simulator or a core implemented in RealView Logic Tiles. Code developed for the ARM926EJ-S Development Chip will also run on the ARM10 and ARM11 processor series.

The expanded system with RealView Logic Tiles can be used to develop AMBA-compatible peripherals and to test ASIC designs. The fast processor core and the peripherals present in the ARM926EJ-S Development Chip, Versatile/PB926EJ-S FPGA, and RealView Logic Tile FPGA enable the user to develop and text complex systems operating at, or near, their target operating frequency.

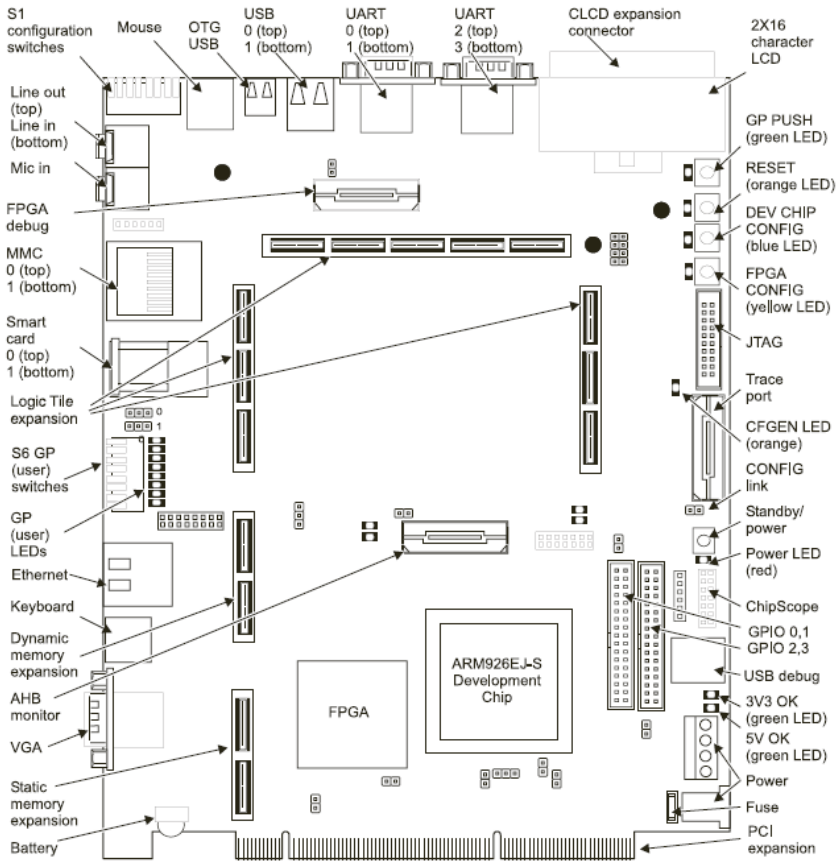


Figure 3.8 : VPB926EJ-S

The following figure shows the architecture of the Versatile PB926EJ-S:

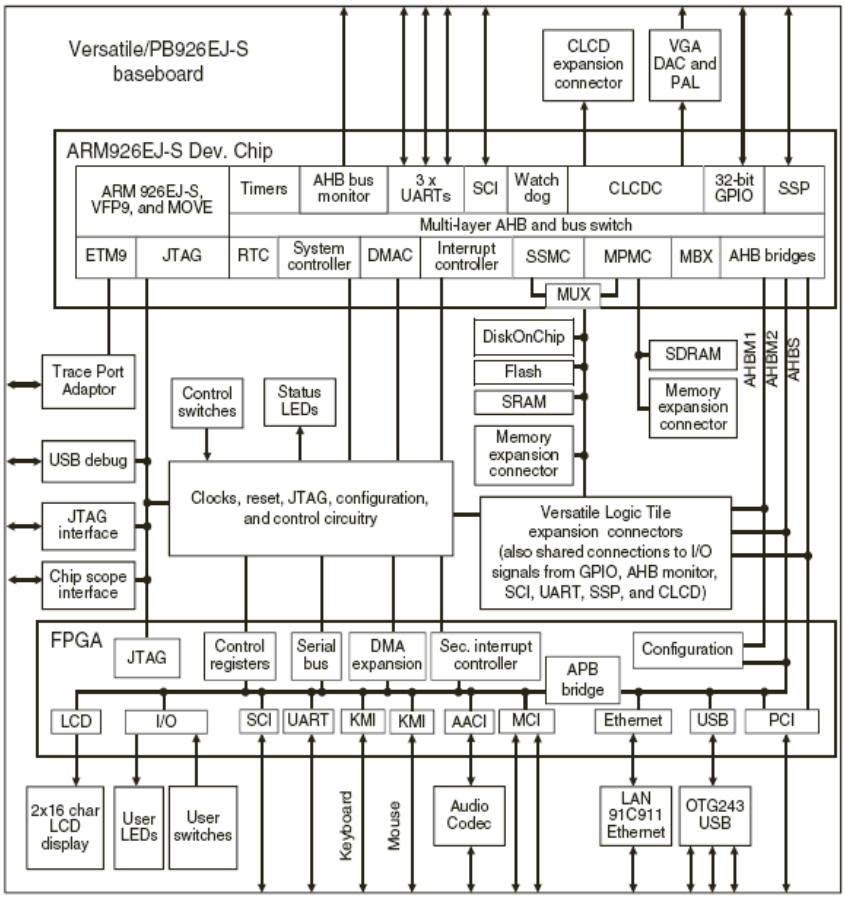


Figure 3.9 : VPB926EJ-S architecture scheme

The Versatile/PB926EJ-S contains the following clock sources:

• crystal oscillators

• five programmable ICS307 clock sources. Two of these are used as the reference for the CPU system clock in the ARM926EJ-S Development Chip and the CLCD controller clock. The other three programmable clocks can be used as external reference clocks for the AHB buses.

• The external clocks of the RealView Logic Tiles can be selected as the reference clocks for theVersatile/PB926EJ-S.

The ARM RealView Logic Tiles enable developing AMBA AHB and APB peripherals, or custom logic, for use with ARM cores. If a RealView Logic Tile is connected, the design in the tile FPGA must implement logic to handle the AHB bus signals. This has been done in our FPGA design with the implementation of the AHB interface at the top level RTL architecture, as explained at chapter 3.1.

AHB S, AHB M1 and AHB M2 are connected to both the FPGA and to the RealView Logic Tile stack. In our system configuration the AHB M1 bus is used for communication between the FPGA CEU Design and the ARM926EJ-S Versatile Board. The FPGA of that Baseboard does not contain any

slaves attached to the AHB M1 bus. The ARM926EJ-S Development Chip memory map assigns the top 2GB of address space (0x80000000–0xFFFFFFFF) to this bus, so the RealView Logic Tile can contain user-supplied slaves that occupy any of this space. In our design the slaves implemented are: AHBTestreg, ahbceureg, CEU_Shell and default_slave (see chapter 3.2 for the memory map). The RealView Logic Tile FPGA must give a response to all transfers that are generated on the AHB M1 bus, even those to addresses in the range 0x00000000–0x7FFFFFFF. The Versatile/PB926EJ-S never generates these addresses on the AHB M1 bus. A separate tile master might, however, generate accesses to this region. But however a second tile is not present in our configuration. It is normal to direct any unwanted transfers to a "default" slave that issues an AHB ERROR response to any active transfers, but a simple zero wait-state OKAY response would be sufficient to ensure that a system functions correctly. That “default” slave has been implemented in our system as already showed in chapter 3.2. If there is not a RealView Logic Tile fitted, pull-up and pull-down resistors on the Versatile/PB926EJ-S ensure that all AHB M1 transfers receive a zero-wait state OK response.

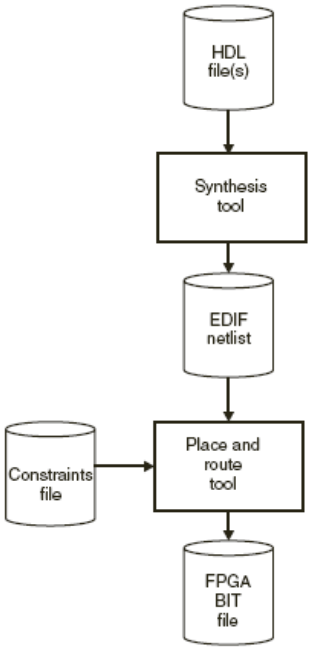
3.5 Design Flow

The flow of my project can be divided in two main steps.

1. The first one concerns the description and simulation of the FPGA Design with a VHDL RTL architecture. That description is followed by its synthesis, and by the FPGA place and route, to finally generate the FPGA bit file.
   This step has been done under Solaris with software tools: Synopsis, Modelsim and ISE.

2. The second step is the one that concerns the hardware environment described in the previous chapters of this fourth session. Its goal is to do a system verification of the IP previously mapped in the Logic Tile FPGA. For this purpose several cryptographic softwares have been written and debugged, with the ARM9 processor, to test the IPDB_CEU behaviour.
   This step has been done under Windows with Lauterbach Debugger and Trace32 Software.

A schematization of the first step has been made in the following figure:



The first part is the production of the HDL files to describe the whole design. The VHDL has been mainly used, but for some mudules both a Verilog and a VHDL description has been adopted. The design is mainly composed of the INFINEON internal IP, that has to be tested, and by an AHB Interface needed to communicate with the ARM Baseboard. Test-benches have also been written for an RTL simulation of all the non pre-existing entities. Modelsim has been utilized for this part of the flow.

Synopsis has then been utilized for the elaboration and compilation of the design in order to produce the EDIF netlist.

That file has been used as input for the ISE software together with a constraint file. Timing constraints have been used for the two clock frequencies of the design : we wanted to use an AHB clock @ 35 MHz and a CEU alternative kernel frequency at 24 MHz. To succeed in having an AHB clock at 35 MHz, we needed to force Synopsis compiler to use specific libraries for the synthesis process. Placement constraints have been used to define the association between the pads of the FPGA and the IO signals. That association is needed because AHB and clock signals are routed from the board to particular pins of the package of the XC2V8000 FPGA.

With ISE the following processes are performed:

- Translate : this is a process where physical constraint and EDIF netlist are analysed in order to detect syntax or logical error.

- Map : The mapping operation is a process where the design is analysed and optimised by removing redundant blocks. A list of the different components that we need for the design is elaborated and a first map of those components on the FPGA is done.
- Place & Route : This process executes the place and route of the design respecting the different constraints. The result of this operation is the creation of an .ncd file. This file can be read using another ISE's tool: FPGA Editor. This tool permits to have a graphical representation of the implementation of the design on the FPGA. By Using FPGA Editor it is possible to modify the place & route and probe some nets. The place and route process also produces timing reports and a post place and route simulation model for the timing analysis.
- Bit generation : after all the precedent processes have been successfully done, it is possible to generate the bit file to program the XC2V8000 FPGA.

For the flow of the second step, chapter 5 will give a detailed description of :

- How to configure the Versatile Platform Baseboard for the ARM9 processor
- How to download the bit output file of the first step to the XILINX FPGA
- How to program the ARM9 processor with the written cryptographic softwares
- How to use the Lauterbach debugger and Trace32 software
- All the functions of the created library for the IPDB CEU
- All the cryptographic programs

4. FPGA Design

The core of the FPGA architecture is the INFINEON's IP designed for cryptographic acceleration: the CEU, which initials in extension mean Cryptographic Encryption Unit. That Unit contains several modules designed for cryptographic applications such as HASH, DES, AES, RSA and a Random Number Generator (RNG). In the beginning I started with a very simple version of the CEU in which there was just the HASH module in which I focused my main works for high level verification. As my works proceeded successfully, I was given new Releases of the CEU to be mapped on the FPGA. Those new releases always had new features and new modules that I could verify on the ARM development board.

The CEU Kernel contains all the main modules intended for cryptographic applications. At a higher hierarchical level, in the CEU_AHB, there are control logics. Those are a module for handling AHB signals (TOPSPIN), a default slave for addresses out of the memory map, and two entities for the SCAN mode. The overall architecture is illustrated in Figure 4.1 of the following paragraph, in which a more detailed description will be done.

My work in INFINEON for the first part of the internship has been to build up a shell for the CEU IP with double-aim of:

1. interfacing new components to the CEU that are needed by its cryptographic modules in the Kernel. Those components have been added in the CEU SHELL entity. They consist of:
    - PRNG: a pseudo random number generator
    - FUSE: a one-time programmable element. (Instantiated multiple times).
    - SRAM: 4 blocks of 128x32bit
2. design an AHB Interface to communicate with the AHB bus of the versatile platform baseboard for ARM926EJ-S. A second slave has been added to map registers in order to monitor and control some FPGA design and CEU IP features.

As INFINEON's CEU IP architecture is confidential, in the following paragraphs only the modules that I instantiated will be illustrated.

4.1 ipdb_ceu_fpga architecture

The following figure illustrates the design hierarchy of the ipdb_ceu_fpga rtl architecture. To make the scheme more readable, just the main blocks and the ones that will be illustrated in the following chapters have been illustrated.

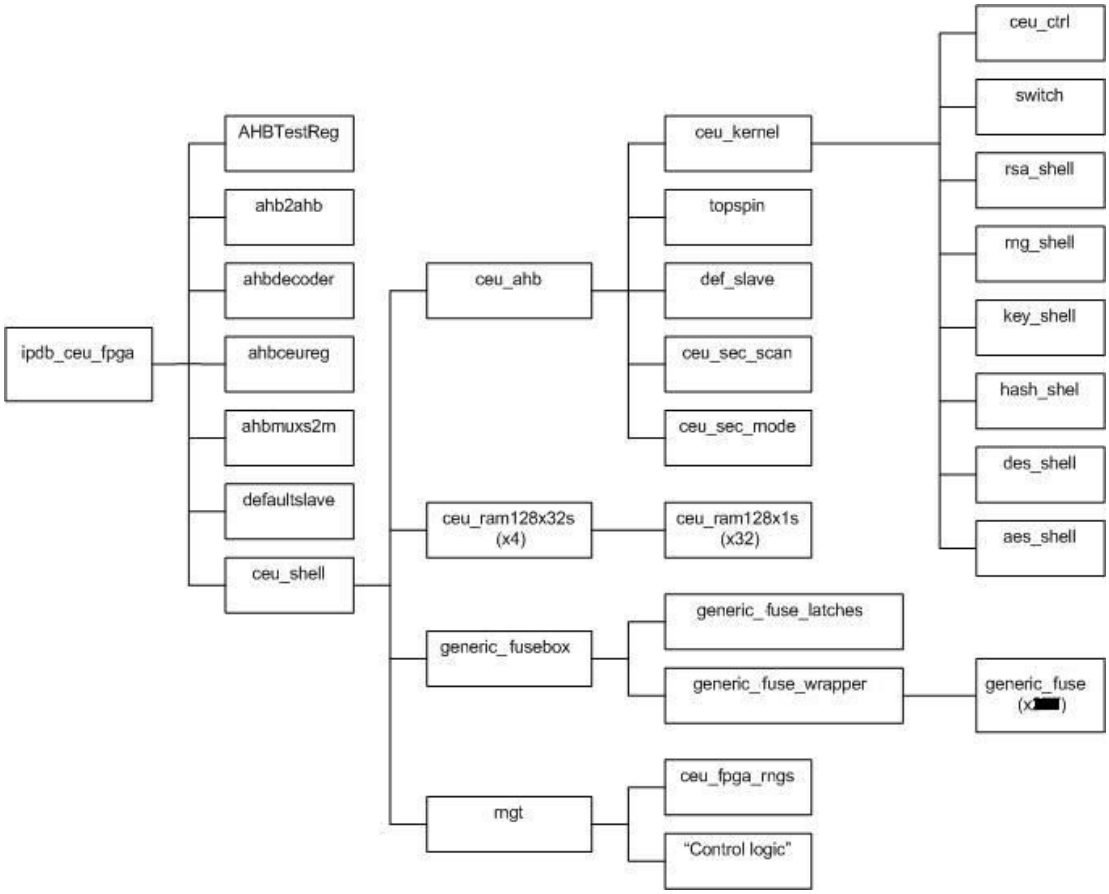
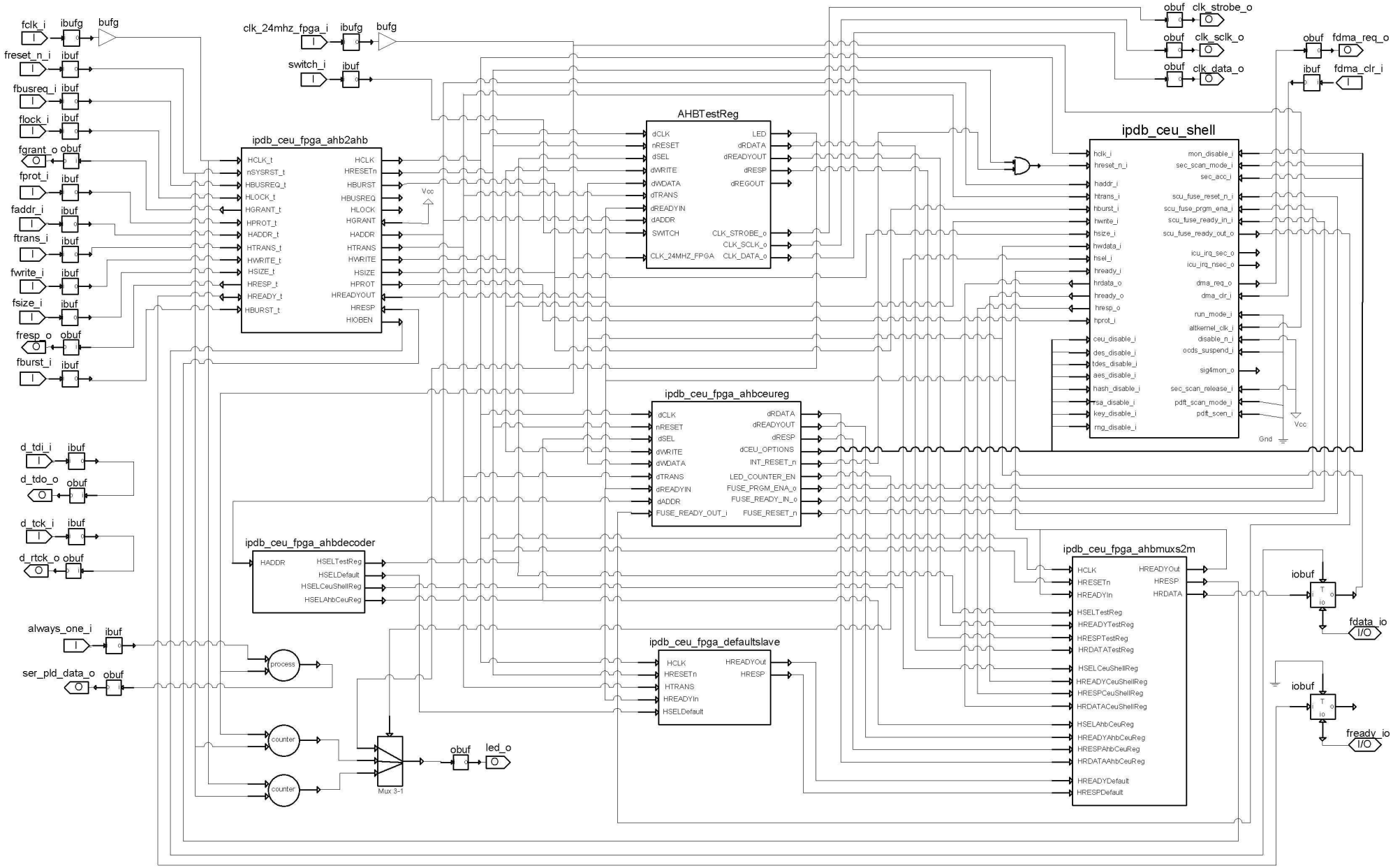


Figure 4.1 : top level design architecture

A detailed schematic representation of the top level RTL architecture is illustrated in the following page. At that top level of the FPGA design there is an instantiation of the Configurable Encryption Unit (CEU) Shell with an AHB interface needed to communicate with the Versatile Platform Baseboard. In addition to the regular AHB interface architecture an ahbceureg entity has been added. That block is used to control and monitor some signal of the CEU Shell; it will be explained in more details in the following chapters. Counters have been implemented just for a test purpose: they are used to check if the clock frequencies are correctly configured. Finally, in that level of the design, FPGA specific pads and clock buffers have been instantiated.

An explanation of each block function follows.

lpdb_ceu_fpga architecture



A simplified architecture with the purpose of explaining the general connection of the blocks is illustrated in the following figure. FPGA specific pads, clock and reset logics have been omitted.

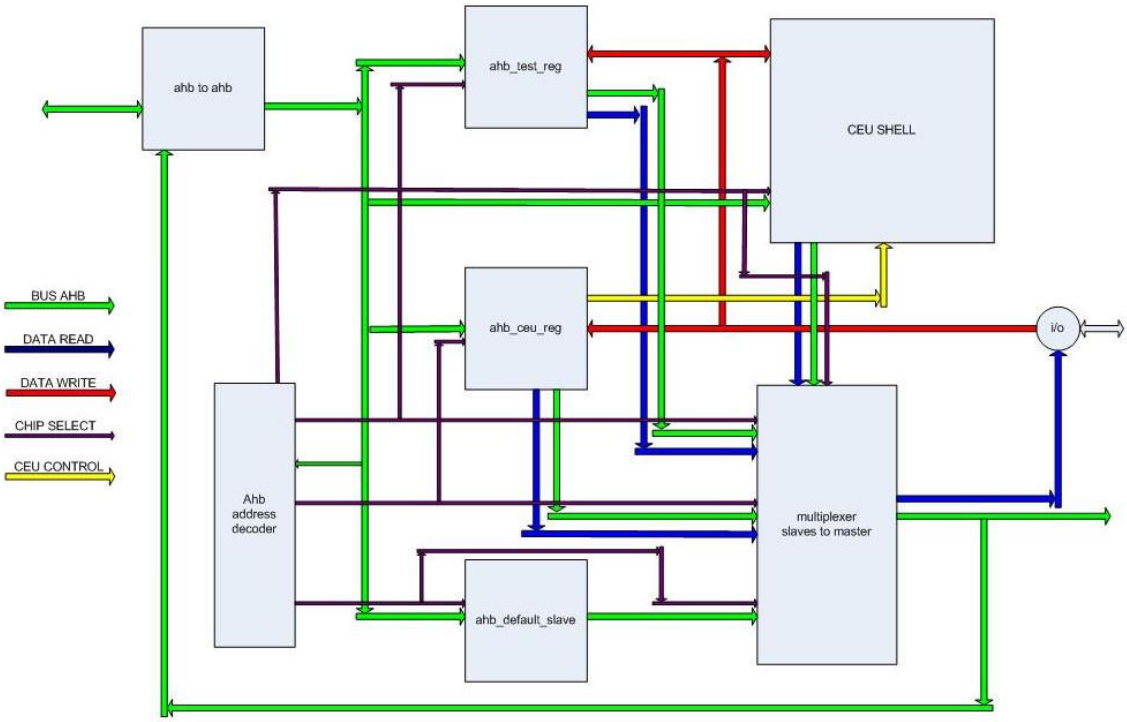


Figure 4.2 : top level design simplified scheme

ahb2ahb : this is the AHB bridge that connect the Versatile Platform Baseboard AHB bus to the AHB signals of the slave modules of the FPGA design.

ahbdecoder : the AHB decoder is used to enable the slave module addressed by the AHB master. Only one slave can be enabled at a time: slave selection is mutually exclusive. The slaves entities of the design are: AHBTestReg, ahbceureg, ceu_shell and defaultslave.

ahbmuxs2m : this block multiplexes the correct input data to the AHB master on the Versatile Board. The slave output data that is selected is the one from the actual slave enabled by the decoder.

AHBTestreg : This slave entity is specific for the architecture of the Versatile Logic Tile XC2V4000+. It has registers and logics that are used for the board programmable clock generators. It also controls the board's switches and leds.

ceu_shell : this slave entity is the core of the design. It contains more than the 99% of the total logic of the design mapped on the FPGA. Inside this entity there is the CEU instantiation together with some CEU external components that are not supposed to be mapped in the ASIC design. Those components are an external Pseudo Random Number Generator, 4 128x32bit SRAMs and a Fusebox. For a more detailed explanation of the CEU Shell refers to the following paragraphs.

ahb_ceu_reg : This slave entity is FPGA design specific for some features of the FPGA logic and for some CEU control and configuration signals. Its description will be explained in details in the following chapters.

default_slave : This slave entity is enabled whenever a out of memory location is addressed. When an undefined area of the memory map is accessed, or an invalid address is driven onto the address bus, the default slave outputs are selected and passed to the current bus master. An OKAY response is generated for IDLE or BUSY transfers to undefined locations, but a two cycle ERROR response is generated if a non-sequential or sequential transfer is attempted.

ibuf, obuf, iobuf : Signals used as inputs to the FPGA must source an input buffer (ibuf) via an external input port. obuf is the output buffer for signals used as outputs from the FPGA. The iobuf is used for bidirectional signals that require that require both an input buffer and a 3-state output buffer with an active High 3-state pin

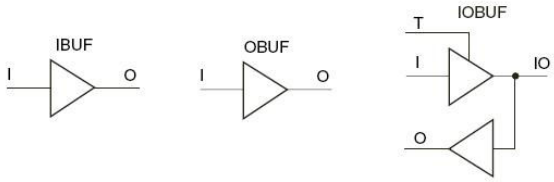


Figure 2.3 : ibuf, obuf, iobuf

ibufg, bufg : This resources are available in the FPGA to manage and distribute the clocks. In the Virtex II FPGA, used on the design, there are 16 ibufg elements that represent the clock inputs.

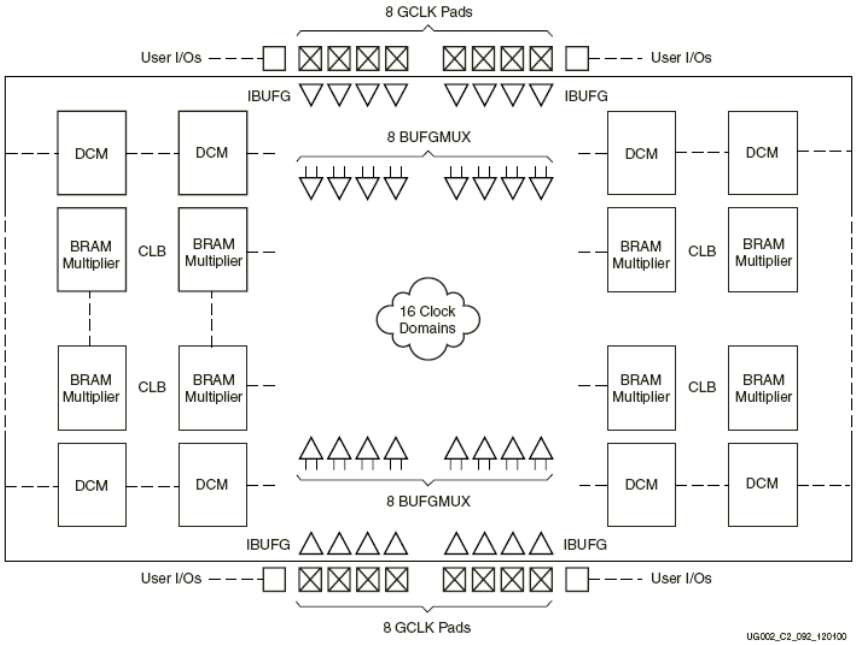


Figure 4.4 : FPGA clock distribution

The simple scheme to distribute an external clock in the device is to implement a clock pad with an ibufg input buffer connected to a bufg global buffer, as shown in Figure 4.5.

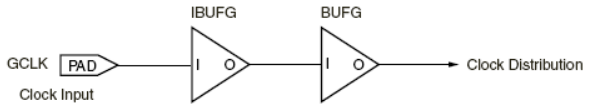


Figure 4.5 : bufg

leds control logic : leds are normally driven by the AHBTestReg via a dedicated register in which 4 bits indicates respectively if the associated led is ON (logic 1) or OFF (logic 0). In the FPGA top level design a multiplexer has been instantiated, to decide if the led output signal is driven by the AHBTestReg or by one of two counters. Those counters are used just for a debug purpose: to test if the two clock frequencies of the design (24 and 35 MHz) are correctly configured. Each counter monitors the cycles of the associated clock and switches the leds after 1 second if the frequency is the one expected. Which of those 3 logics has to drive the leds (AHBTestReg, 24Mhz counter, 35Mhz counter) is decided by the ahb_ceu_reg entity.

4.2 Memory mapping

The AHB M1 memory mapping is as follows:

AHB M1 Slave	Address range	Dedicated Registers	Description*
Default	0x 0000 0000 0x 7FFF FFFF	-	V/PB926EJ-S System memory map
Test Registers	0x 8000 0000 0x 8FFF FFFF	+ 0x00 : FPGAID	Versioning of Implemented hardware (RO)
		+ 0x04 : LTSWITCH	Provides LT Switches states (RO)
		+ 0x08 : LTLEDS	Controls LT Leds state (RW)
		+ 0x0C : TEST_REG	Provides Register output for test (RW)
		+ 0x10 : OSC0	Controls Programmable Clock Arbiter (RW)
		+ 0x14 : OSC1	Controls Programmable Clock Arbiter (RW)
		+ 0x18 : OSC2	Controls Programmable Clock Arbiter (RW)
		+ 0x1C > 0xFF	Unused
Default	0x 9000 0000 0x EFFF FFFF	-	-
FPGA Registers	0x F000 0000 0x F3FF FFFF	+ 0x00 : FPGA_STAT	Status register : fusebox
		+ 0x04 : FUSE1	Fuse word 1
		+ 0x08 : FUSE2	Fuse word 2
		+ 0x0C : FUSE3	Fuse word 3
		+ 0x10 : FUSE4	Fuse word 4
		+ 0x14 : FPGA_CFG	CEU Export Restriction and Security Options (RW)
		+ 0x18 : FPGA_CTRL	Software Reset, Led Counter driver and Fuse Programming (RW)
		+ 0x18 > 0x20	Unused
CEU Shell Registers	0x F400 0000 0x F5FF FFFF	Confidential information	
Default	0x F600 0000 0x FFFF FFFF	-	-

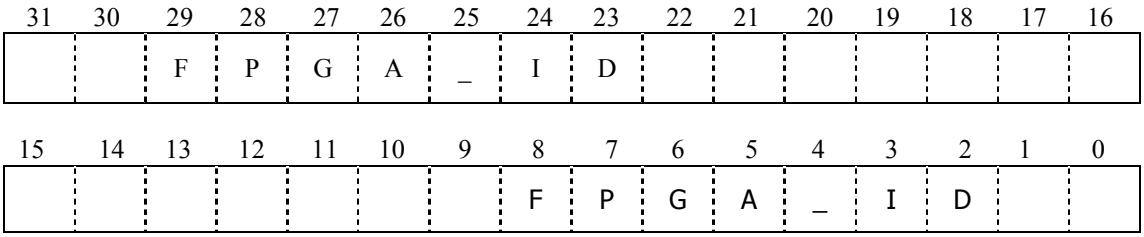
*RO : Read Only. RW : Read Write

4.3 Test Registers Definition

4.3.1 FPGA Identification Register

This status register is only readable.

FPGA_ID 0x80000000 [Reset value : depends on hardware version]

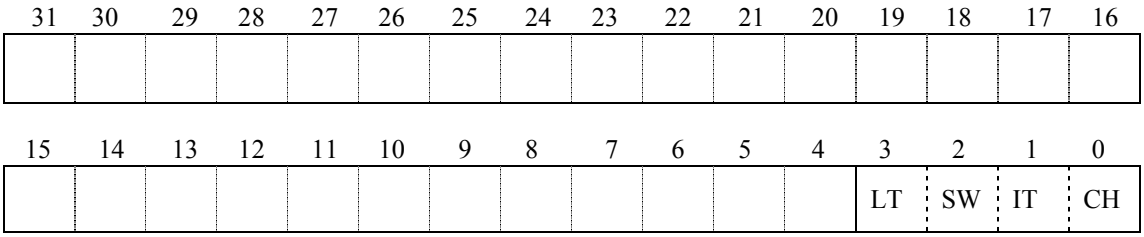


Field	Bits	Typ	Description
FPGA_ID	[31:0]	RO	Versioning of Implemented FPGA hardware

4.3.2 Logic Tile Switch Register

This status register is only readable.

LTSWITCH 0x80000004 [Reset value: depends on LT switches state]

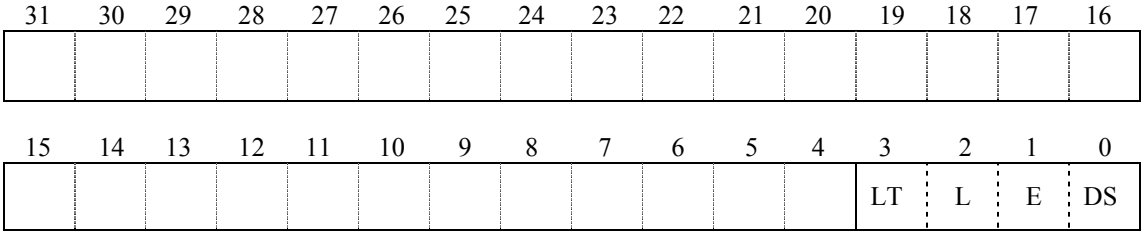


Field	Bits	Typ	Description
LTSWITCH	[3:0]	RO	Provides the 4 LT Switches states 0 LT Switch off. 1 LT Switch on.

4.3.3 Logic Tile Leds Register

This configuration register is always readable and writable.

LTLEDS 0x80000008 [Reset value: 0x00000000]

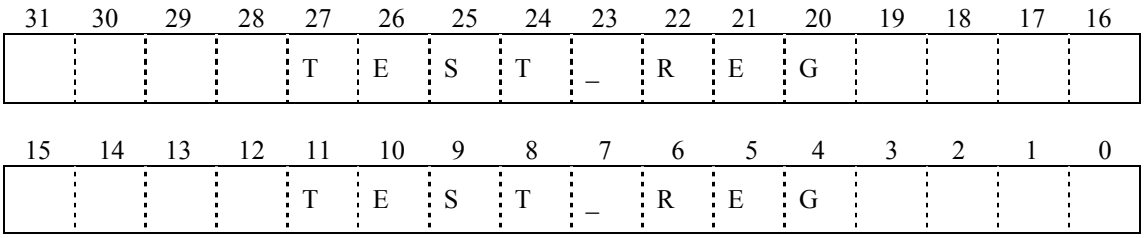


Field	Bits	Typ	Description
LTLEDS	[3:0]	RW	Controls the 4 LT Leds states 0 LT Led off. 1 LT Led on.

4.3.4 Test Register

This register is always readable and writable.

TEST_REG 0x8000000C [Reset value: 0x00000000]



Field	Bits	Typ	Description
TEST_REG	[31:0]	RW	Test register for write and read accesses

4.3.5 Logic Tile ICS307 Oscillators

This configuration register is always readable and writable. For a detailed description on Serially Programmable Clock Source, please refer to “ICS307-01/02” Datasheet.

OSC0	0x80000010	[Reset value : 0x00020281]
OSC1	0x80000014	[Reset value : 0x00030406]
OSC2	0x80000018	[Reset value : 0x00030406]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
						-							S2	S1	S0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V8	V7	V6	V5	V4	V3	V2	V1	V0	R6	R5	R4	R3	R2	R1	R0

Field	Bits	Typ	Description
S	[18:16]	RW	Output Divider Select (OD)
V	[15:7]	RW	VCO Divider Word (VDW)
R	[6:0]	RW	Reference Divider Word (RDW)

4.4 FPGA Registers Definition

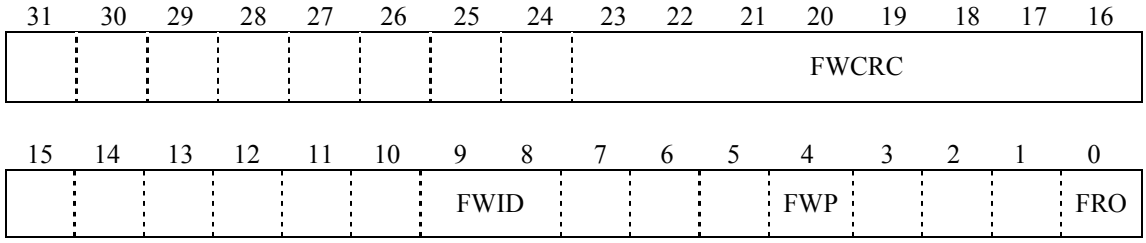
Those Register are mapped on the ahb_ceu_reg VHDL unit.

4.4.1 FPGA Status Register

Fuse Ready Out (FRO) is set high at the end of the e-fuse sensing or programming operation. *Fuse Word CRC* (FWCRC), *Fuse Word ID* (FWID) and *Fuse Word Protection* (FWP) are all values read from the Fusebox. Only half of the fuses are read (including the protection value), as the other half is a redundancy copy for errors checking.

This status register is always only readable.

FPGA_STAT 0xF0000000 [Reset value: 0x00XX0XXX]



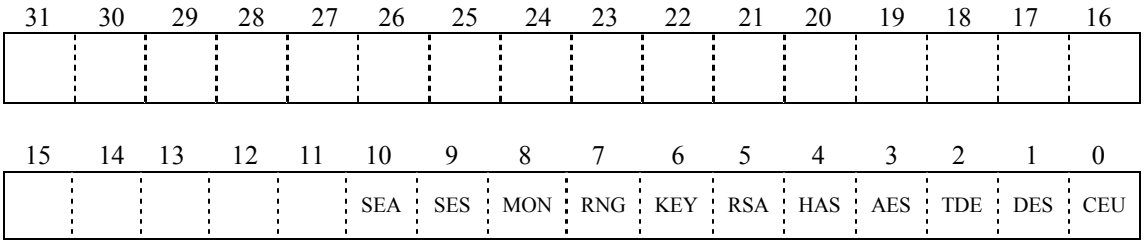
Field	Bits	Typ	Description
FWCRC	[23:16]	RO	**Fuse Word Computed CRC**
FWID	[9: 8]	RO	**Fuse Word ID :** **0-** an external source was used as word originator **10** the internal "X" was used in fixed-compression mode. The quality of the word is unknown **11** Secure Fusing Mode. The internal "X" was used in quality mode. The quality of the word is securely high
FWP	[4]	RO	**Fuse Word Protection** **0** fuses can be programmed **1** fuses programmed. Reprogramming is prevented
FRO	[0]	RO	**Fuse sensing or programming operation finished** **0** no operation started or finished **1** operation finished.

4.4.2 FPGA Configuration Register

This register drives the configuration input pins of the IPDB CEU which are provided in order to statically configure the CEU block. In this way it is possible to selectively disable any cryptographic algorithm (disabling its module) without compromising the overall CEU functionality.

This configuration register is always readable and writable.

FPGA_CFG 0xF0000014 [Reset value: 0x00000400]



Field	Bits	Typ	Description
SEA	[10]	RW	**Chip security system settings.** This is a control signal intended to be driven by a system secure-access control mechanism. **0** A non-secure access is executed. The behaviour of the CEU depends on the internal CEU mode. **1** A secure access is executed. All CEU Registers are accessible.
SES	[9]	RW	**Enables secure scan mode. See CEU implementation Specs for details** **0** Disable **1** Enable
MON	[8]	RW	**Disables all monitoring signals, so it is not possible to extract HW monitoring information** **0** Enable **1** Disable
RNG	[7]	RW	**Disables any access to the RNG generator, if present** **0** Enable **1** Disable
"X"	[6]	RW	**Disables any access to the "X" Manager sub-module, if present** **0** Enable **1** Disable
RSA	[5]	RW	**Disables any access to the RSA accelerator and its RAM, if present** **0** Enable **1** Disable
HAS	[4]	RW	**Disables the full Hash sub-module, if present** **0** Enable **1** Disable
AES	[3]	RW	**Disables the full AES sub-module, if present** **0** Enable **1** Disable
TDE	[2]	RW	**Disables only the 3-DES algorithm within the DES sub-module, if present** **0** Enable **1** Disable
DES	[1]	RW	**Disables the full DES sub-module, if present (both DES and 3-DES algorithm** **0** Enable **1** Disable
CEU	[0]	RW	**Completely disables any CEU access. All bus accesses return an ERR response** **0** Enable **1** Disable

4.4.3 FPGA Control Register

This register can be used to force the reset signal to the CEU. When the software reset is active the CEU will stay in reset mode ignoring all other external signals. Otherwise the CEU will work normally and its reset signal will be driven by the external system reset input pin: *freset_n_i*. That pin is connected with the push button of the Logic Tile.

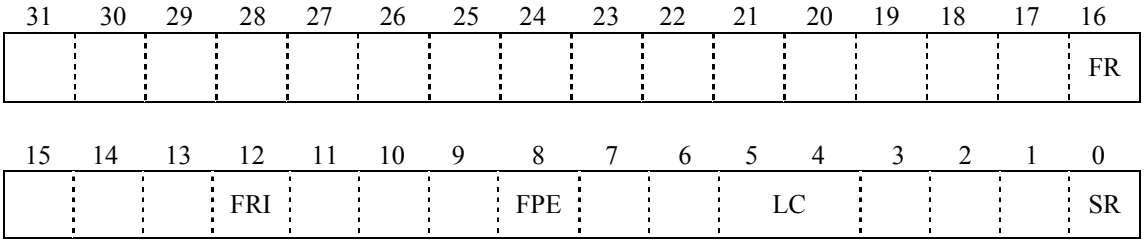
This register is also used to select if the toplevel output signal *led_o*, which light the leds on the Logic Tile, is driven by a counter or by the AHBTestReg module. The counters are used to flash the leds every second to check if the clock frequencies (24Mhz and 35Mhz) on the Logic Tile are correct.

Fuse sensing or programming procedure can be controlled by the FPE and FRI bits. *Fuse Programming enable* (FPE) enables the e-fuse programming. Both fuse programming and sensing procedures can only start when the *Fuse Ready In* (FRI) is high. FRI is set high after global reset.

EFR is the *Fuse Reset* control signal. After a global reset this bit is set high, if it is programmed to logic '0' all the fuses registers are cleared.

This control register is always readable and writable.

FPGA_CTRL 0xF0000018 [Reset value: 0x00011001]



Field	Bits	Typ	Description
SR	[0]	RW	**This bit force the CEU to reset** **0** CEU forced to reset status **1** CEU works normally.
LC	[5: 4]	RW	**This bit select if the leds on the Tile are driven by the AHBTestReg or by an internal counter** **0x** LTLeds driven by AHBTestReg **10** LTLeds flash every second @ 35Mhz ahbclk **11** LTLeds flash every second @ 24Mhz ceuclk
FPE	[8]	RW	**Fuse programming enable** **0** disable **1** enable
FRI	[12]	RW	**Fuse ready for sensing or programming** **0** not ready **1** ready
FR	[16]	RW	**This bit is used to reset the Fuses RTL decription** **0** Fuses reset to logical zeros **1** Fuses can be programmed

4.5 CEU Shell Architecture

The internal architecture of the ipdb_ceu_shell is illustrated in the following figure.

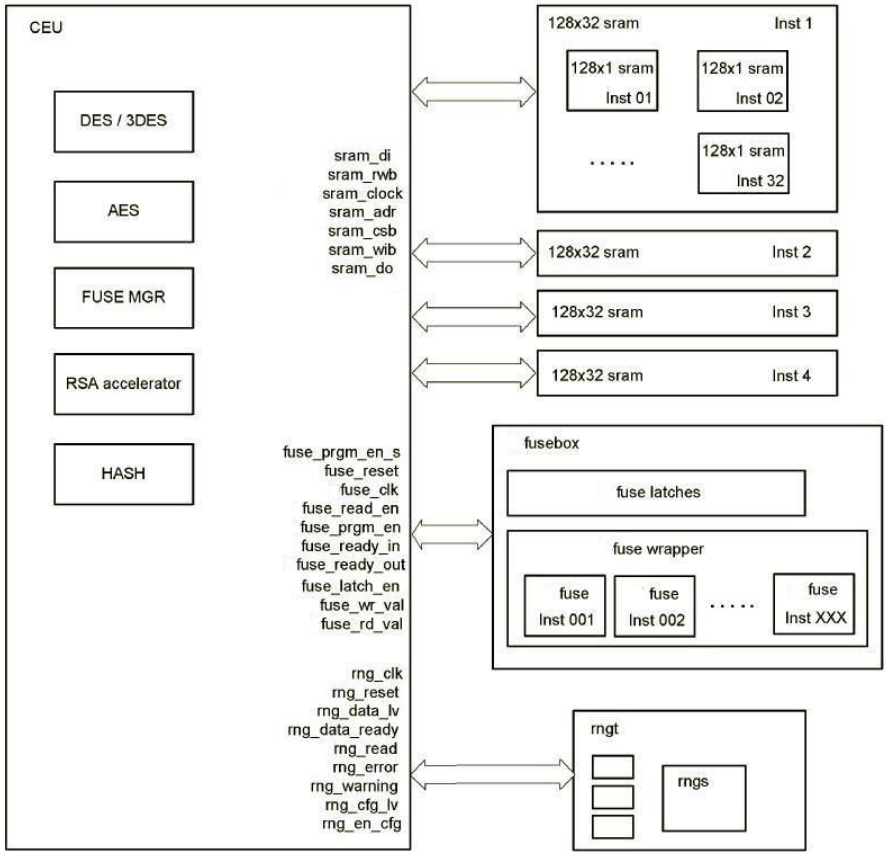


Figure 4.6 : CEU Shell architecture

From that figure it is easy to see that the architecture of the Shell is an instantiation of the IPDB_CEU together with some external components that are not intended to be mapped in the ASIC design. The “fusebox”, “SRAM”, and “RNG” entities instantiated in the FPGA design are an RTL description of the real components that are supposed to be integrated off-chip in the overall ASIC design. They are intended for a debugging purpose of the overall system that has to be mapped in the FPGA. Those RTL architectures are a simplified description of the pre-existing behavioural ones. Yet Figure 4.6 is not to be considered a complete explanation of the CEU Shell, because not all the sub-entities and not all the signals are showed.

A more detailed explanation of each external component architecture follows :

4.5.1 RNG

The top level of this structure is the rngt entity, which is described in Verilog. The rngs was the pre-existing entity which was used to describe, with a behavioural architecture, the random number generation process. This entity has been replaced with a VHDL RTL architecture named

"ipdb_ceu_fpga_rngs". Every time that random 32 bits numbers are requested by the CEU to the rngt, the rngt activates and configures the rngs, which is supposed to be in power down mode, and reads its the serial 1 bit pseudo random output. A ready signal is activated when the 32 bit data is ready, and new data is generated when a read signal is asserted in response. A test bench is available in the ipdb_ceu_shell folder under Solaris for random data generation at the rngt level and ipdb_ceu_fpga_rngs design level.

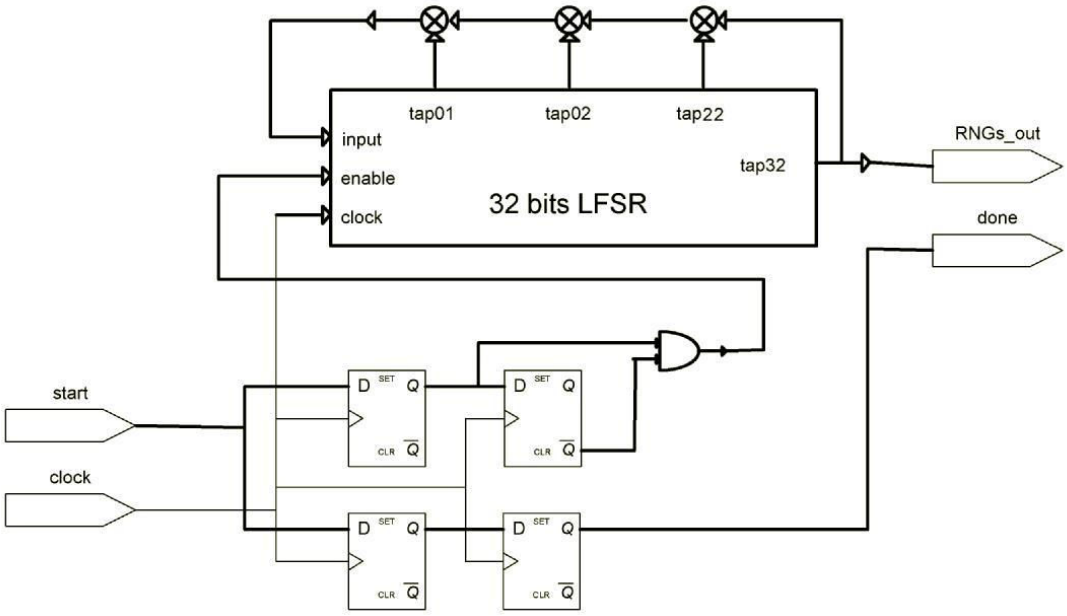


Figure 4.7 : CEU Shell RNG architecture

The ipdb_ceu_fpga_rngs has been implemented with a 32 bits Linear Feedback Shift Registers, as showed in Figure 4.7. Even if the rngt control logic is intended for a more sophisticated sub-module, the ipdb_ceu_fpga_rngs is designed with such an interface that it is compatible with the design. The LFSR generates, as known, pseudo random data. The outputs of the flips flops of the LFSR are XNORed in feedback to obtain a maximum length counter of $2^{32}-1$. To do so the outputs must be taken from the flip-flops instantiation numbers 1, 2, 22 and 32. At system reset the LFSR flip-flops are

configured to a randomly chosen state. This is done with their reset and set synchronous inputs which are not showed in Figure 4.7. An n-bit maximum sequence length LFSR counter goes through all possible code permutations except one, which would be a lock-up state. The XNOR feedback makes the lock-up state the all-ones state. The XOR would make the lock-up state the all-zeros state.

Every time that random data is needed, the rngt generates a round square to the start input pin of the rngs. The start signal is sent to a single clock cycle impulse generator which is composed by the two upper flip-flops with the outputs connected to the and-gate. This impulse is used to enable the flip-flops of the LFSR for one cycle so that it evolves to its next state and outputs the bit of its last tap to the RNGS_out. Those serial outputs are used to generate the 32 bits data by the rngt. The done signal is just a 2 clock cycles delay of the start input.

4.5.2 FUSEBOX

Fuses are one-time programmable elements. The architecture of this IP is confidential.

4.5.3 RSA-SRAM

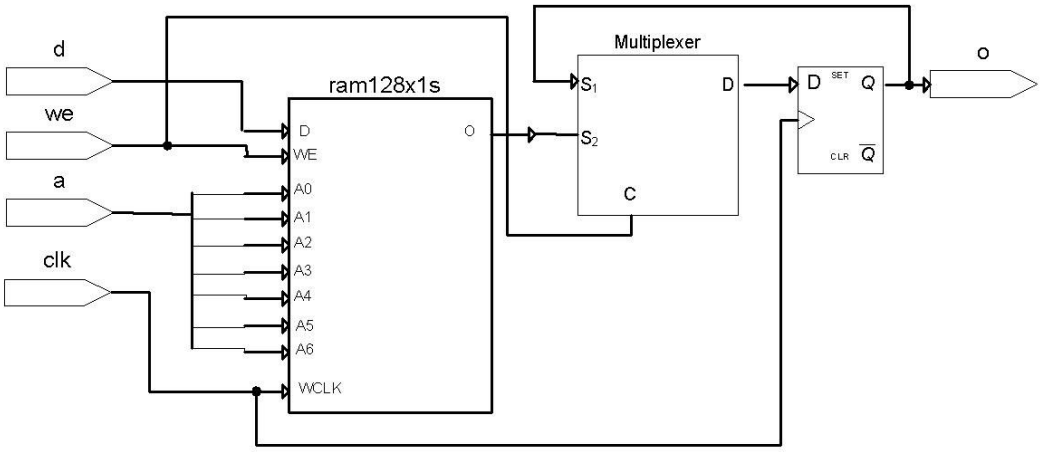


Figure 4.8 : CEU Shell 128x1 SRAM block

4 blocks of 128x32 bits SRAM are needed by the RSA sub-module of the IPDB_CEU. To create a SRAM of that dimension a smaller block of RAM, provided by the XILINX UNISIM library, is instantiated multiple times.

The cell block utilized is the Distributed SelectRAM Memory named ram128x1s. That block is showed in Figure 4.8. That memory writes synchronously and reads asynchronously. The write operation is a single clock-edge operation, with a write enable that is active High by default. When the write enable is Low, no data is written into the RAM. When the write enable is High, the clock edge latches the write address and writes the data on D into the RAM. The read operation is a combinatorial operation. The address port is asynchronous with an access time equivalent to the logic delay. When new data is synchronously written, the output reflects the data in the memory cell addressed (transparent mode).

Two features of those RAM block need to be avoided because of the architecture of our design: as showed in Figure 4.8, the following logic has been added to do so:

- As the read operation is asynchronous, it has been synchronized with a D flip-flop sampling the output.
- The transparent mode behaviour at writes operations has been avoided with the insertion of multiplexer. When the write enable is active, then the output of the flip-flop is in feedback with its input, to keep its previous state. In read mode the feedback loop is broken and the D flip-flop samples the output of the ram128x1s block.

5. Software Use Cases

In this last chapter will follow a description of the software written in C code for an application oriented verification of INFINEON's CEU IP mapped on the FPGA. The main aim of those programs is to make an high level system verification in order to monitor the behaviour of the CEU kernel cryptographic modules in various applications. Those applications include interaction of several modules and the use of hardware specific for software cooperation. The debugging application utilized with the LAUTERBACH Debugger, connected on the JTAG port of the VPB926EJ-s, is TRACE32.

HASH :

As the first Release of the CEU had only the HASH Module enabled, this one was the first verification target. INFINEON needs to Hash a big block of data from the Boot ROM in order to authenticate it. This is the situation previously described in the HMAC paragraph (see 2.5.4 example d). In order to hash it, this data has to split in multiple sub-blocks. Each sub-block has to be hashed atomically, but between each sub-block there may be an interruption to run a different application. For this kind of scenario I developed some functions that I stored in a library; in a second time I did some modification to those functions in order to make them compatible also for different scenarios.

In order to hash a message split in multiple packets, I developed a function to hash a single packet, which output the intermediate digest, and a function to initialize the hash sub-module with the intermediate digest from the previous hash. I also wrote a function to hash a whole message: this function splits the message in a number of packets that the user can set, and calls the function for packet hashing as many times as needed. Other utility functions have also been written.

HMAC:

CEU hardware logic supports the Hash sub-module in executing the Hashed Message Authentication Code (HMAC) algorithm. That algorithm is illustrated in Figure 5.1, where: K0 is the Secret Key, ipad and opad are two different constant strings stored in the CEU, text is the message, || is the appending operation, and H() is the hash function. The operation of XORing the message with the ipad and opad strings is always done by the hardware. CEU supports two different algorithms for executing the HMAC: The Hardware Flow and the Software flow.

Hardware HMAC flow: with this kind of flow the algorithm is completely handled by hardware, the software that I wrote just sends to the Hash sub-module first the Key and then the message. The hardware handles all the steps; of course with the exception of the last step in which the user can decide how many bytes of the final digest will compose the HMAC. With this flow the message has to hashed with the stand alone configuration: whole message has to be hashed atomically (without interruptions). To do so, the hash packet from the library functions can be used.

Software HMAC flow: this flow is intended for application switch support. In case a big block of data has to be hashed, the message can be split in multiple packets. As explained in the Hash section, each packet has to be hashed atomically, but between each packet a context switch is allowed: the intermediate digest saved from the last hashed packet will be used to initialize the Hash sub-module when the next packet will be taken care. Of course each packet transmission is handled by the software. The transmission of the massage is the step 5 in Figure 5.1.

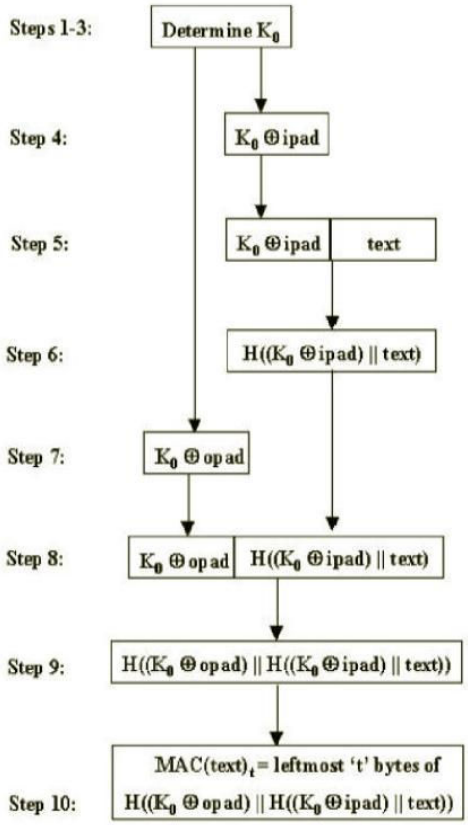


Figure 5.1 : HMAC algorithm scheme

With this flow the software has to take care of more steps than the hardware flow, but we have the advantage of the context switch support. The software has to handle the key transmission in step 4 and 7, and the message transmission in step 5 (multiple packet sending) and step 8 (single packet sending). For all those steps the hash packet function from the CEU library is utilized.

The software and the functions will be described in details in the following paragraphs.

AES WRAPPED DES KEY:

This a particular application in which we use a secret key to unwrap (decrypt) a second key that we use to encrypt the message. The secret key is already supposed to be known by the CEU, we use it to handle a second key which is encrypted and has then no information for someone who would catch it. This flow is typically used for Digital Rights Management (DRM).

This flow has been utilized to verify the correct behaviour of the system while multiple CEU sub-modules are interacting between them. The data follow multiple paths in the CEU internal architecture.

Those 3 software use cases are probably the most important and more interesting between the one that I wrote. Other functions of the library and other applications will be described in the following paragraphs. A description of how to set up the debugging environment and how to handle data has been done in the Appendix.

5.1 Functions from the CEU Library

There will follow a short description of all the functions, that I wrote, used in the cryptographic source codes. Information about those functions is mainly confidential, because they access IPDB_CEU internal registers. Just a general description, when possible, will be done.

void clean_sdram (uint32_t start, uint32_t end) : this function initialize the SDRAM memory writing all zeros from the <start> address to the <end> address (<end> address excluded). It can be used only to clean addresses from 0x04000000 to 0x08000000 which are reserved for the cryptographic programs data storage.

void conf_mode() : confidential

void run_mode() : confidential

void fifo_tx (uint32_t base_address, uint32_t lwords) : This function is used for Data transmission from the SDRAM of the Versatile Board to the IPDB_CEU in the FPGA design.

void iv_data_send (uint32_t md_length, uint32_t ceu_hash_cfg) : This function is used to configure and set the initialization value of the HASH module when a multiple packets hashing is performed (when the plain text is split in multiple blocks). The intermediate digests are read from the dedicated SDRAM memory.

void save_digest (uint32_t md_length) : This function saves, in the SDRAM memory area reserved for intermediate digests, <md_length> words from the "X" CEU reception register. "X" is polled to wait for Data to be ready in "X" register. Every time that this function is called the old saved value is replaced by the new one.

void store_final_digest (uint32_t digest_lwords, uint32_t msg_i, uint32_t data_dg_offset) : this function stores, at the <msg_i> location of the SDRAM area reserved for final digests, <digest_lwords> words from the memory area reserved for intermediate digests. That means that every time that the last packet of a message has been hashed (or the whole message if the message has not been split), the intermediate digest previously saved by the save_digest() function becomes the final one and then is copied also to the memory area reserved for final digests. Every time that this function is called it doesn't replace the previously written values if a different value for <msg_i> is always used. This function also checks if the CEU computed digest is the same of the one stored at <data_dg_offset> SDRAM memory offset. If so it writes a word with all zeros just after the last digest stored word, otherwise it writes a word with all ones. <data_dg_offset> memory offset is loaded with the digests from the TRACE32 input binary file if all the previously described procedures are followed.

If the final digest is not known before the CEU hashing, the digest_check word can be ignored without compromising other procedures.

void hash_packet (uint32_t packet_address, uint32_t hash_words, uint32_t hash_cfg, uint32_t sw_byte_cnt) : This function is used for packet hashing. It can be used to hash a single block of a message split in multiple packets and also to hash a single packet message. <packet_address> is the starting memory address in which the block to be hashed is stored. <hash_words> is the number of 32bit words of the packet. <hash_cfg> is the configuration value for the Hash Module. <sw_byte_cnt> is intended for context switch support application. It is of particular need in the software HMAC flow cryptographic source code.

This function utilizes the VPB926EJS DMA Controller to send the message packet from the SDRAM to the IPDB_CEU. The DMA controller is configured with the "memory to peripheral with peripheral as flow controller" option, where the peripheral is a IPDB_CEU module. The burst size is of 16 words of 32 bits. Interrupt specific registers are configured to allow the communication with the DMA. At the end of the hash_packet() function the save_digest() is called.

void hash_msg (uint32_t algo, uint32_t msg_address, uint32_t msg_bytes, uint32_t packet_bytes) : This function is used to do the hash of a message of any length (for the maximum message length refers to CEU architecture). This function, according to the input values, can split the message in multiple packets and launch the hash_packet() procedure a number of times equal to the number of blocks in which the message has been split. Otherwise a stand-alone hash of the message is executed launching the hash_packet() just once. Every time that the hash_packet() is invoked a proper hash_cfg is passed, according to the packet being hashed and the total number of packets.

Between the arguments of the hash_msg() function, <algo> refers to the Hash algorithm that has to be performed. <msg_address> refers to the starting memory address in which the message is stored. <msg_bytes> is the total length of the message expressed in bytes. <packet_bytes> is the length in bytes of the packet in which the message has to be split. If this last value is greater or equal than <msg_bytes>, then a stand-alone hash is executed. Otherwise the message will be split in a number of packets equal to the integer quotient, rounded by excess, of <msg_bytes> divided by <packet_bytes>.

If a stand-alone hash is executed then the Hash Module is configured properly.

If the message is split in multiple packets just after having hashed the first packet, before launching again each hash_packet(), a iv_data_send() is executed to initialize the Hash module with the intermediate digest result of the previous packet.

At the end of the hash_msg() it is always suggested to launch a store_final_digest() to avoid the final result being overwritten by a new hashing procedure.

5.2 Cryptographic Source codes

An explanation of all the written cryptographic programs, that I wrote, follows. Most information about those programs is confidential, because they access IPDB_CEU internal registers and sub-modules. Just a general description, when possible, will be done.

Hash of known test vectors with DMAC support : This source code do the hashes of a test vector which is loaded in the SDRAM data memory from the binary input file generated following the procedure explained in the "Data conversion for TRACE32 managing" chapter. In this way the first information in the data memory is the total number of messages to be hashed, then it will follow periodically information on "message length", "message" and "digest". With the "message length" information, as the digest has a fixed length dependent from the hash algorithm, the code can easily jump from a message block data to the next one, starting from the first one which has a constant address. This program then just invokes the hash_msg(), always followed by a store_final_digest(), a number of times equal to the number of messages to be hashed. Data extraction is also explained in the "Data conversion for TRACE32 managing" chapter.

In the "Data" subfolder there are sha-1 and sha-256 test vector, both for short and long messages. For each of them there is already the associated input binary file and the output reports. Following the explained procedure, or just editing those input files, it easy to create new test vectors.

HMAC with Hardware Flow : This source code performs the HMAC of a single message using the Hardware control flow. It configures the IPDB_CEU for the HMAC key, then sends the key and waits for HMAC initialization to be done. It finally does a stand-alone hash with the hash_packet() function.

The source code is ready to be used with the "hwhmacex.bi" as input binary data from TRACE32. Doing so it will execute the following example:

```
hex key =      0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
key_len =      20 bytes
ascii data =   "Hi There"
hex data =     0x4869205468657265
data_len =     8 bytes
hex digest =   0xb617318655057264e28bc0b6fb378c8ef146be00
```

To run any other test with different data, one can follow a similar procedure:

- Write to SDRAM starting form address 0x04100000 first the message, ...
- ... then leave a space equal to the final digest length (it depends from the algorithm) ...
- ... then write the key.
- at line 60 of the code set the message algorithm
- at line 61 set the packet length (in this test case set it bigger than the message)
- at line 90 set the message length in bits
- at line 99 set the key length in bits

HMAC with Software flow, whole message hashed : This source code does the HMAC of a single message using the Software control flow. The message is not split in multiple packets. The flow of this algorithm can be resumed in four steps:

1. The Hash Module is configured for the Key. A hash_packet() is launched with the key values as arguments (the key is simply treated as it was a message).
2. The Hash Module is then configured in "X" mode. A hash_packet() is launched with the message data as argument. A store_final_digest() is also executed, cause we will need this value in step 4, but it will be previously overwritten by step 3.
3. The Hash Module is configured for the Key. A hash_packet() is launched with the key values as arguments (the key is simply treated as it was a message).
4. The Hash Module is then configured in "X" mode. A hash_packet() is launched passing, as argument, the digest stored in step 2. A store_final_digest() is once again executed, overwriting the intermediate digest of step 2 with the final digest computed by the hash_packet() of this last step.

In the source code folder there is already a binary file to be loaded with TRACE32 to execute the following example which utilizes the sha-1 algorithm for the hashing:

Key:
00010203 04050607 08090a0b 0c0d0e0f
10111213 14151617 18191a1b 1c1d1e1f
20212223 24252627 28292a2b 2c2d2e2f
30313233 34353637 38393a3b 3c3d3e3f

(Key xor ipad)||text:
36373435 32333031 3e3f3c3d 3a3b3839
26272425 22232021 2e2f2c2d 2a2b2829
16171415 12131011 1e1f1c1d 1a1b1819
06070405 02030001 0e0f0c0d 0a0b0809
53616d70 6c652023 31

(Key xor opad) || Hash((Key xor ipad)||text):
5c5d5e5f 58595a5b 54555657 50515253
4c4d4e4f 48494a4b 44454647 40414243
7c7d7e7f 78797a7b 74757677 70717273
6c6d6e6f 68696a6b 64656667 60616263
bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2
7b7e1b20

HMAC(Key, Text) = Hash((Key xor opad) || Hash((Key xor ipad)||text)):
4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d
a0403c0a

To run any other test just follow the steps explained at the beginning of the C source file. The instructions are quite the same of the ones explained in the hardware HMAC test case, they just differ for the line position in the code.

HMAC with Software flow, multiple packet hashing for context switch support : This source code does the software HMAC of a single message. The message is split in 3 packets. The flow of this example is quite the same of the one in the "sw_hmac_ex1_singlepack" source code, the only two differences are that and additional pre-step is needed to hash the key and a multiple packets hashing is performed at the second step.

- Key hashing : This additional step must be executed at the very start of the flow of the program as the key is 1216 bits long. The key is hashed, the digest result is stored in the SDRAM with the store_final_digest() function, and finally the old key address is updated with this new one. The new length of the key is equal to the length of the digest of the utilized hash algorithm.
- Multiple packets hashing : Step 2 is performed with a multiple packets hashing. The flow for hashing a message split in blocks is similar to the one explained in the hash_msg() function. This flow is intended for applications requiring context switch support and can be of particular use when a large data block to be hashed is split over multiple packets.

Loading the "swhmacex2.bi" binary file as input data in TRACE32, the following example is executed:

```
Key =     aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
          aaaaaa                                (131 bytes)
Data =    546869732069732061207465737420 75     ("This is a test u")
          73696e672061206c6172676572207468       ("sing a larger th")
          616e20626c6f636b2d73697a65206b65       ("an block-size ke")
          7920616e642061206c61726765722074       ("y and a larger t")
          68616e20626c6f636b2d73697a652064       ("han block-size d")
          6174612e20546865206b6579206e6565       ("ata. The key nee")
          647320746f2062652068617368656420       ("ds to be hashed ")
          6265666f7265206265696e6720757365       ("before being use")
          642062792074686520484d414320616c       ("d by the HMAC al")
          676f726974686d2e                       ("gorithm.")

HMAC-SHA-256 = 9b09ffa71b942fcb27635fbcd5b0e944
               bfdc63644f0713938a7f51535c3a35e2
```

Test of the PRNG : This source code just enables the CEU_Shell RNG, outputs 256 words of 32 bits from the RNG, and stores the data in the SDRAM.

Test of the SDRAMs : This source code saves in the CEU_Shell SRAMs the first 16 words from the VPB926EJS SDRAM memory reserved for the input data from TRACE32. Then reads and saves back to the board memory, in a different area, the first 20 words from the CEU_Shell SRAMs. Last 4 words should be all zeros.

AES Encryption : This source code encrypts a single message with the AES Module in Electronic Code Book (ECB) Operation Mode. The flow is explained in 3 steps :

1. "X".
2. The AES Module is configured for ECB encryption, and then the Key is configured. "X" is polled to wait that the AES Key is successfully loaded and ready for use.

3. The message can now be sent to the AES. The encrypted message in output is saved to SDRAM.

In the program directory there is already a binary file named “ECBex1.bi”. If loaded with Trace32, the following example can be automatically executed:

Key 2b7e151628aed2a6abf7158809cf4f3c
Plaintext 6bc1bee22e409f96e93d7e117393172a
Ciphertext 3ad77bb40d7a3660a89ecaf32466ef97

To run any test just load the memory from address 0x04100000 with the message hex value, followed by the key hex value. Set line 167 with Decryption or Encryption setting. Change line 56 with correct message length value in bits. Change line 62 with correct key length value in bits.

AES wrapped DES key, with trailing zeros : This source code runs a DES ECB message encryption with a key that has to be previously unwrapped with the AES Module. The purpose of this program is to the test a key unwrapping scheme of the DES/3DES key. This flow is typically used in Digital Rights Management (DRM). A particular environment is created: we want to load a 64 bit DES key managing a 128 bit padded key with the AES Module. This can be done if the correct half of the padded key, the one carrying information, is loaded as DES key1; and the other half, the one carrying the zeros padding, is loaded as DES key2 and then ignored in the encryption. The flow is here explained in 9 steps:

1. the AES Module is configured for decryption in ECB mode
2. the key is loaded to the AES
3. the data path is configured.
4. the DES Module is configured for encryption in ECB mode
5. the data size is configured
6. the wrapped key is sent from the SDRAM to the AES. This unwrap the key and load the decrypted DES key to the DES.
7. the data path is configured.
8. The message is sent to the DES
9. The encrypted message in output is checked

If the binary file of the same directory of the source code is loaded to the SDRAM with TRACE32, the following example is automatically executed:

128 bit Key (AES) [0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f]

64 bit RAW DES KEY [0x13,0x34,0x57,0x79,0x9b,0xbc,0xdf,0xf1]

RAW DES Key Padded with Trailing Zeros
RAW_DES_KEY_TZ [0x13,0x34,0x57,0x79,0x9b,0xbc,0xdf,0xf1,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00]

Wrapped Des Key padded with trailing zeros
WR_DES_KEY_TZ [0xb4,0x6f,0xfe,0xb6,0xce,0xab,0x6a,0x61,0xd4,0x18,0xb4,0x79,0x16,0xe7,0x10,0x0f]

```
 Sample plain text for DES encryption in ECB mode (16 byte text)
PLAIN_TEXT              [ 0xf8,0xcf,0xac,0x1a,0x1b,0xde,0x2f,0xca,0xf3,0xc8,0x0d,0xf5,0x19,0x1b,0x59,0xbe ]

Plain Text Encrypted with RAW_DES_KEY
ENC_TEXT                [ 0x87,0xcb,0x97,0x2d,0x98,0xe6,0x94,0x4e,0x26,0x40,0x14,0x42,0x3c,0x70,0x4d,0x05 ]
```

AES wrapped DES key, with leading zeros : This code is quite the same of the aes-wrapped_des_key_tz one. The only difference is that the 64 bits DES key is padded with leading zeros instead of trailing zeros: that means that this time the second half of the 128 bits AES key is the one carrying the information. If we would follow the same flow of the other code, the zeros padding first half would be loaded as DES key1 and used for encryption. To avoid this we need one more step in the flow to swap the two half of the key. This can be done with a configuration of the CEU modules with a different word order sequence for input and output. The flow is here explained in 10 steps:

1. the AES Module is configured for decryption in ECB mode. Word order sequence is configured.
2. the key is loaded to the AES
3. the data path is configured.
4. the DES Module is configured for encryption in ECB mode. Word order sequence is configured.
5. the data size is configured
6. the wrapped key is sent from the SDRAM to the AES. This unwrap the key and load the decrypted DES key to the DES.
7. The word order sequence is configured.
8. the data path is configured.
9. The message is sent to the DES
10. The encrypted message in output is checked

If the binary file of the same directory of the source code is loaded to the SDRAM with TRACE32, the following example is automatically executed:

```
Raw DES Key padded with Leading Zeros
RAW_DES_KEY_LZ          [ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x13,0x34,0x57,0x79,0x9b,0xbc,0xdf,0xf1 ]

Wrapped Des Key padded with Leading Zeros
WR_DES_KEY_LZ       [ 0xeb,0xeb,0x6d,0xdf,0x7d,0x43,0x55,0xcd,0xb1,0xf2,0x21,0x65,0xf4,0xf9,0x96,0x5d ]
```

6. Conclusions

The Software Debugged on the ARM Versatile Board where the FPGA, with the Infineon's IP mapped on it, was connected, has succeeded in doing an application oriented verification of the Cryptographic Encryption Unit (CEU). The FPGA Prototyping has permitted to validate the CEU architecture in particular for some use cases with hardware/software co-verification. The hardware allows in fact particular software flows. One of those particular application, for example, has been explained in the Hardware/Software HMAC introduction of chapter 5.

Our testing platform is currently being utilized by the Component Verification Team. The cryptographic functions that I wrote will provide a base for the developing of software drivers for INFINEON's CEU IP.

My 6 months stage in Infineon Technologies France has been a really positive experience for my formation. I learned a different flow approach to the VHDL simulation and synthesis steps with Modelsim and Synopsys Tools, and the use of ISE for the Place and Route. I enriched my knowledge working with XILINX VIRTEX II FPGA, AMBA Bus, ARM9 Processor, LAUTERBACH Debugger, CLEARCASE Data Managing System. More than all, I had the opportunity to participate to a digital front-end design flow with an expert team specialized in the field.

I also would like to thanks Infineon's equip for this to be possible and the Design Department for having followed me during my internship.

Appendices

Appendix A: Setting Up the Versatile Platform

Configuration switches S1 and S6, shown in Figure A.1, control how the Versatile/PB926EJ-S configures itself and the action to take after reset.

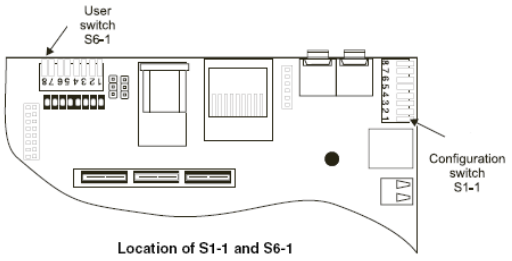


Figure A.1

The configuration switches Sl-1 to SI-8 determine boot memory type, the FPGA image, and the RealView Logic Tile image, memory configuration, and FPGA options at power on. If the switch lever is down, the switch is ON. The default is OFF, switch lever up. The following table explains each switch function.

Switch	Default	Function in default position
S1-1 and S1-2	OFF	Selects Disk-on-Chip as boot memory
S1-3	OFF	Selects synchronous AHB bridge mode.
S1-4	OFF	Reserved (SSMC enabled), leave in OFF position
S1-5	OFF	Selects OSCCLK frequency of 35MHz.
S1-6 and S1-7	OFF	Selects Versatile/PB926EJ-S FPGA image 0
S1-8	OFF	Selects RealView Logic Tile FPGA image 0

Figure A.2 : switches functionality

All the switches are left in their default position, the only exception is S1-3 which is set ON to select asynchronous AHB bridge mode.
Switches S6-1 and S6-3 are used for the Boot Monitor, the remaining switches in S6 are available for user applications. All these switches are left in their default position.

The Jumper J32 (next to the power up/down button) must be ON, to put the board in configuration mode. In this way it is possible to program the FPGA, if connected to the board. In normal mode (jumper J32 OFF) the FPGA loads its configuration data from a flash memory device. In that mode it is possible to program the ARM processor, to run it, and to debug it with Trace32 PC Software.

Appendix B: Hierarchical Directories Structure

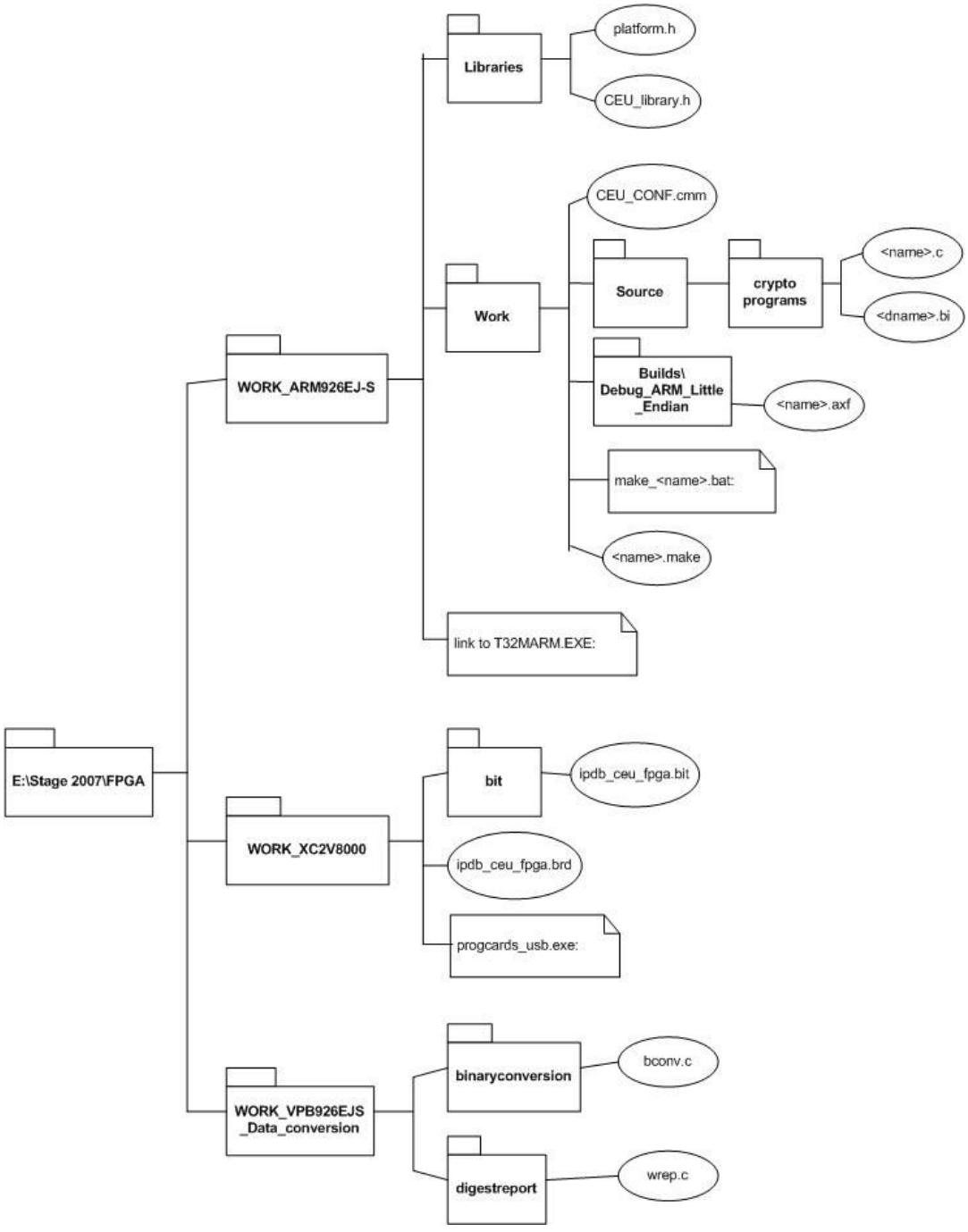


Figure B.1 : Hierarchical directories structure

WORK_XC2V8000

This directory is used to map the FPGA Design in the Xilinx XC2V8000 FPGA mounted on the XC2V4000+ Logic Tile. The bit file, generated by ISE after the Place & Route has been completed, has to be placed in the bit directory. The template.brd file in the WORKXC2V8000 directory has to be

renamed and edited with the name of the bit file. It is suggested to give them the same name keeping of course the different extension. The progcards_usb.exe application will start the programming of the FPGA asking the name of the brd file script to launch.

WORK_ARM926EJ-S

This directory is used to program the ARM processor in the Versatile Board. In the Source directory there are few crypto program folders containing the C codes and their data for TRACE32 inputting if needed. Every C source code includes two libraries: “platform.h”, which defines all the board constants and utility functions, and the “CEU_library.h”, which defines all the ipdb_ceu_fpga constants and also few utility and crypto functions. For each C source code there are two associated files in the “Work” directory: a .make script and a .bat executable. This last file can be launched to compile its associated C crypto program. This operation generates, in the “Debug_ARM_Little_Endian” directory, a main.axf image, which is overwritten every time that any code is compiled, and a .axf image with the same name of the associated C file. That last image is overwritten only when its C source code is compiled. With the shortcut to T32ARM.exe TRACE32 software can be launched. If then “T32_ARM926EJ-s_CEU_CONF.cmm” batchfile is loaded, TRACE32 will ask to the user which data binary file (.bi) and code image (.axf) are to be used for debugging.

WORK_VPB926EJS_Data_conversion

This directory contains two programs used under Solaris to convert the crypto input and output data in a format that can be easily handled with TRACE32. "bconv.c" is used to convert an ASCII text file, containing HASH test vectors, in a binary file that can be loaded to the Board SDRAM so that it can be computed by the ARM processor. "wrep.c" examines the hexadecimal output file, extracted from the Board SDRAM memory after having executed the HASH test vector code, and generates a text report that can be easily read by the user.

The following chapters describe, in a more detailed way, how to download FPGA BIT files into the XC2V8000 FPGA using Progcards application, and how to modify and download the crypto programs intended for the ARM926EJ-S processor.

Appendix C: FPGA Bit File Download

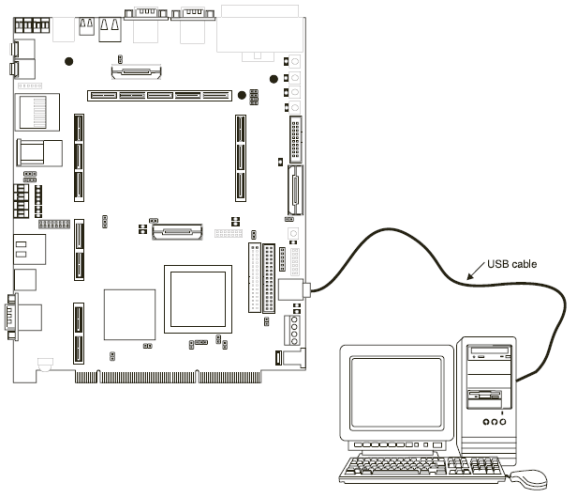


Figure C.1 : FPGA bit file download

- You may download an existing version of bit file into the FPGA or create your own.

- Once a Bit file has been created from the design flow, copy it to the following directory: "E:\Stage2007\FPGA\WORK_XC2V8000\bit".

- Change the file template.brd in "E:\Stage 2007\FPGA\WORK_XC2V8000".
  <Instance_name> : name of the instance you want to appear in the download file selection using Progcards.
  <file_name>.bit : name of the bit file to download.
  Save the file to a new name (*.brd)

- Connect the USB debug cable directly to the USB Debug port on the mother board.
  Board's drivers are needed

- Place the configuration jumper (J32).

- Switch on the board: LEDS D12 to D20 should be flickering.

- Launch Progcards_USB : "E:\Stage 2007\FPGA\WORK_XC2V8000\progcards_usb.exe".

- Select the bit file to download.

- The program exits once the bit file is properly downloaded.

- Switch off the board and remove the configuration jumper (J32)

Appendix D: Set Up the TRACE32 Debugger

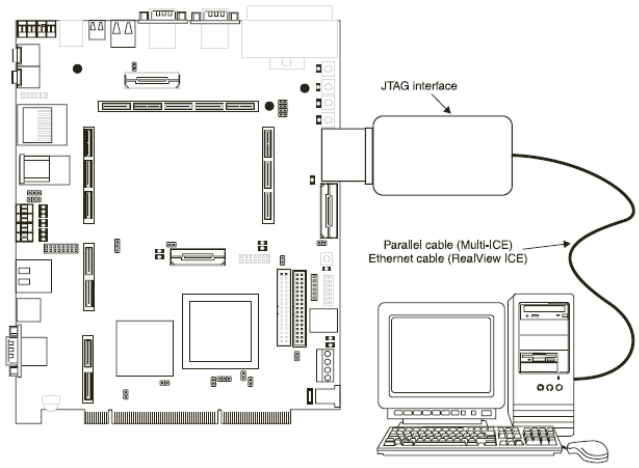


Figure D.1 : FPGA debug with LAUTERBACH

Use Trace32 debugger to download and execute the program on the ARM processor. This emulator & configures the development board, downloads the image file of the program intended for the ARM processor, configures the user environment of the debugger and automates the generation of hundreds of configurations. In order to configure and use the IPDB CEU FPGA manually, please refer to the following steps:

- Connect the USB debug cable to the USB debug interface from LAUTERBACH.
- Connect the JTAG adapter intended for ARM9 to the USB debug interface from LAUTERBACH.
- Connect the JTAG cable to the JTAG connector on the board (J31).
- Switch on the board.
- Launch TRACE32 emulator &debugger.
- Once ARM CPU family is detected, select "Batchfile..." in the "File" menu to download the configuration file of the VPB for ARM926EJ-S. *
- Select the configuration file (*.cmm) you created or the following one for a manual use of the IPDB CEU FPGA:
  "E:\Stage2007\FPGA\WORK_ARM926EJ-S\Work\T32_ARM926EJS_CEU_CONF.cmm".
  If that batchfile is loaded, TRACE32 will ask for a data binary file (.bi) and a code image (.axf). If data is not needed just skip it.
- Download the image file (*.axf) you created.

Appendix E: TRACE32 Debugger

After having launched TRACE32 debugger, if the CEU configuration batchfile is loaded as previously explained, this is how the graphic interface will appear:

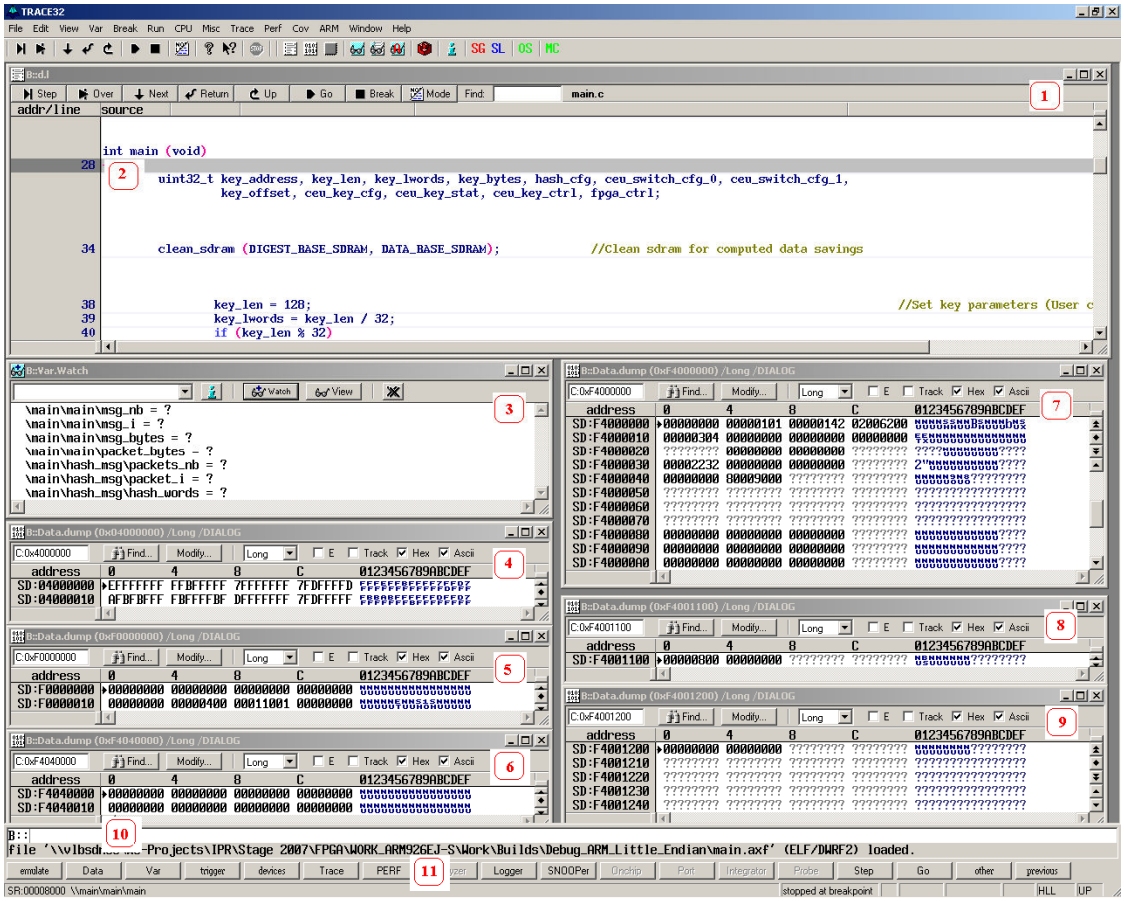


Figure E.1 : TRACE32 Graphic User Interface

Area 1 : This window shows the C code of the program which is running on the ARM processor. Remember to respect the directory hierarchy and the steps previously explained otherwise it may happen that you won't be able to see any code in this window or, even worse, that the code won't be consistent with the one on the ARM processor. By default the code is shown in HLL Mode: this means that only C code will be shown and each instruction line can be debugged once at a time. It is possible to switch to MIX Mode also: under each C code line its translation in Assembler will be displayed. To do so click from the menus "Run – Mode (F9)". It will be possible to debug each assembler instruction at a time.

To run a single step debug instruction click on "Run – Step (F2)". Doing so it's possible to watch each register behaviour after each instruction. To run the whole program just click "Run – Go (F7)". After a "Run – Break (F8)" the program will stop to run and the registers status will be updated. It could be useful to place breakpoints: "Break – Set" or just double click on the desired line in the code of Area 1. If a "Go" instruction is given to TRACE32, the program will run and stop at the

breakpoint position. This is useful to use for the debugging the clean_sdram() function, present in all the crypto programs, which otherwise would take several debugging single step instructions.

Area 2 : The gray background line shows the next instruction that will be debugged with a single step.

Area 3 : In this window it is possible to watch, and also modify, all the variables of the programmed source code. To modify a variable double click on it and enter the new value. It is possible to modify variables in the middle of a program but only if the processor is not running: that means that it is always possible to modify a variable between each single step debug instruction (F2). To add variables click on the "watch" tab. A hierarchical menu, with all the functions of the actual source code and their variables, will appear.

Areas from 4 to 9 are all the same kind of window: they show the status of the memory of the board. It is possible to modify the memory just as described for the variables in Area 3. This is very useful to control the FPGA memory mapped registers and the Versatile Board registers and peripherals . To add a new window of this kind click from the menus "View - Dump" and set the address. Or just modify the address of an already existing window.

Area 4 : Data saving reserved Sdram memory. All the crypto programs store in this memory area the intermediate digest results. All the final results are placed starting at 0x100 offset address from here.

Area 5 : AHB/FPGA Registers.

Area 6 : CEU Transmission register.

Area 7 : CEU General registers, Basic Interrupts and DMA Registers.

Area 8 : CEU DES-Specific Registers.

Area 9 : CEU AES-Specific Registers.

Area 10 : This line is used to give commands to TRACE32 Software. For a list of available TRACE32 commands with their explanation you can refer to the following guide "T:\T32\MAN.HLP".

Area 11 : This area shows a list of the possible commands that can be inputted in Area 10 in respect of the language structure rules. For example if in Area 10 a "data." command has been written, in this area tabs such as "view", "load", "save", "copy", "print" and "set" will be displayed, meaning that a "data.load." is one of the possible accepted commands. This area could be very useful for a beginner TRACE32 programmer.

To modify the graphic structure and the options of Trace32 and the Versatile Board settings it is suggested to create a new configuration file from the already existing one : T32_ARM926EJS_CEU_CONF.cmm.

After having loaded the batchfile "T32_ARM926EJS_CEU_CONF.cmm" the user will be prompted which binary data has to be loaded to the memory. Every crypto subfolder in the Source directory contains a main code and, if it is needed, its binary associated data. After the program has been ran, and the final digest computed and saved to memory, it is possible to save it to an output text file with the following command:

DATA.SAVE.ASCIIHEXB "<file_path_name>" SD:<start_mem_address>--SD:<end_mem_address>

Usually the <start_mem_address> where the computed data is saved is 0x04000100, the <end_mem_address> depends from the program.

Appendix F: Data Conversion for TRACE32 Managing

The two C codes under the folders "binaryconversion" and "digestreport" are used to manage data for input and from output with TRACE32. The HASH program with which this particular data format can be used is located in the "Source\2007_10_03_dma_hash_test_vector" directory. This code do the hash of a number n of messages of any length.

TRACE32 input data : The first program "binaryconversion\bconv.c" converts an ASCII text file, containing *n* HASH vectors, in a binary file that can be easily loaded to the Versatile Board SDRAM memory. ASCII characters are converted to their hexadecimal format and bytes composing each 32 bit word are swapped for an easy management with the memory boundary. The text file has to respect a particular structure to be successfully converted and then analyzed by the crypto code loaded in the ARM processor. The message structure is the following one:

Len = <message_length_in_bits_using_5_chars>

Msg = <hexadecimal_message>

Md = <hexadecimal_digest>

This structure has to be utilized for each message to be hashed and can be repeated as many times as needed. The conversion program will add the number of messages information as the first 32bit word of the binary file. Message length, even if expressed in bits, must always be a multiple of byte because of the CEU architecture.

The code needs a constant to be changed for hashing with the sha-1 or sha-256 algorithm. The constant defines the digest length expressed in number of hexadecimal characters and is located at the first line of the main() function. It must be "const int lengthmd_hex = 64;" for sha-256, and "'const int lengthmdhex = 40;" for sha-1.

This is the procedure to be followed for input data conversion:

- Place the "bconv.c" text code under a Unix directory
- Enable Gcc with the command "module load gcc"
- Compile the program with the command "gcc bconv.c -o bconv"
- Name the hash test vector file as "testvector.txt"
- Launch the command "bconv > report.txt".
- The needed binary file will be saved with the name "binary.bi", rename it before launching the command for another test vector. Copy it under Windows.

- To load the binary file to Trace32, launch it and configure it with the CEU_CONF batchfile. The program will ask you to select the binary file to load to memory automatically. Otherwise you can use the following command in TRACE32:
  "data.load.binary <file_path_name> <memory_address>"

TRACE32 output data : The second program "digestreport\wrep.c" converts an ASCII file, written from TRACE32 and containing the hexadecimal values stored in the SDRAM memory, in a second ASCII text file with a more friendly format. The values to be converted are the digest results from the hashing of the input test-vector. Each digest is followed by the word "0x00000000" if the CEU computed digest passed the check with the digest reported in the input data, by the word "OxFFFFFFFF"' otherwise.

The code needs two constants to be changed for hashing with the sha-1 or sha-256 algorithm. At code line 32 use "if (j == 5)" for sha-1 and "if (j == 8)" for sha-256. At code line 47 use "if (j == 6)" for sha-1 and "if (j == 9)" for sha-256. This values refer to the number of 32bit words composing the digest of each algorithm.

This is the procedure to be followed for output data conversion:

- From TRACE32, after having ran the hash program in "Source\2007_10_03_dma_hash test-vector"' directory, save the data from the memory to an output text file. To do so use on of the following commands :
  for SHA-256 use "data.save.asciihexb <file_path_name>.txt SD:04000100--SD:04000A00"
  for SHA-1 use "data.save.asciihexb <file path name>.txt SD:04000100--SD:04000700"
  Modify the SDRAM end address in the previous commands if you are using a test-vector different from the ones already present in the source hash directory.

- Copy the output file you want to convert in a Unix directory and rename it "mem_output.txt"

- Place the "wrep.c" text code under the same Unix directory

- Enable Gcc with the command "module load gcc"

- Compile the program with the command "gcc wrep.c -o wrep"

- Launch the command "wrep > file_name.txt"

An example: this is an example of data conversion for sha-1:

- Input ASCII test-vector before conversion to binary format:

```
Len = 00008
Msg = a8
MD = 99f2aa95e36f95c2acb0eaf23998f030638f3f15

Len = 00016
Msg = 3000
MD = f944dcd635f9801f7ac90a407fbc479964dec024

Len = 00024
Msg = 42749e
MD = a444319e9b6cc1e8464c511ec0969c37d6bb2619

Len = 00032
Msg = 9fc3fe08
MD = 16a0ff84fcc156fd5d3ca3a744f20a232d172253
```

- Output ASCII Hex file from TRACE32 before wrep conversion:

```
95 AA F2 99 C2 95 6F E3 F2 EA B0 AC 30 F0 98 39
15 3F 8F 63 00 00 00 00 D6 DC 44 F9 1F 80 F9 35
40 0A C9 7A 99 47 BC 7F 24 C0 DE 64 00 00 00 00
9E 31 44 A4 E8 C1 6C 9B 1E 51 4C 46 37 9C 96 C0
19 26 BB D6 00 00 00 00 84 FF A0 16 FD 56 C1 FC
A7 A3 3C 5D 23 0A F2 44 53 22 17 2D 00 00 00 00
```

Note: while the memory is watched under TRACE32 it will appear in this other boundary:

```
99 F2 AA 95 E3 6F 95 C2 AC B0 EA F2 39 98 F0 30
```

but the bytes in the 32bit words are swapped by TRACE32 while writing them to the output text file. That is the reason why a swap procedure has been implemented a second time in the data conversion "wrep.c" code.

- Output file from TRACE32 after wrep conversion:

```
il digest del messaggio n. 1 e': 99F2AA95 E36F95C2 ACB0EAF2 3998F030 638F3F15    OK

il digest del messaggio n. 2 e': F944DCD6 35F9801F 7AC90A40 7FBC4799 64DEC024    OK

il digest del messaggio n. 3 e': A444319E 9B6CC1E8 464C511E C0969C37 D6BB2619    OK

il digest del messaggio n. 4 e': 16A0FF84 FCC156FD 5D3CA3A7 44F20A23 2D172253    OK
```

Bibliography

Thomas Lueftener, Joerg Berthold, Christian Pacha, “A 90-nm CMOS Low-Power GSM/EDGE Multimedia-Enhanced Baseband Processor With 380-MHz ARM926 Core and Mixed-Signal Extensions”. IEE Journal of solid-state circuits, vol.42, NO.1, January 2007

OMTP Hardware Working Group, “Trusted Environment OMTP TR0, Definition and Requirements”

William Stallings, “Cryptography and Network Security, Principles and Practices”. Prentice Hall, fourth edition

ARM DUI 0224B, “Versatile Platform Baseboard for ARM926EJ-S. User Guide”. ARM

ARM DUI 0186B, “Versatile™/LT-XC2V4000+ Logic Tile. User Guide”. ARM

XILINX DS031, “Virtex-II Platform FPGAs: Complete Data Sheet”. XILINX

XILINX UG012, “Virtex-II Pro and Virtex-II Pro X FPGA User Guide”. XILINX

FIPS PUB 198, “The Keyed Hash Message Authentication Code (HMAC)”.